Бинарные деревья поиска

- **Двоичное T-дерево *упорядочено*, если для любой вершины X справедливо свойство:**

все элементы, хранимые в левом поддереве, меньше элемента, хранимого в X,

а все элементы, хранимые в правом поддереве, больше элемента, хранимого в X.

Бинарные деревья поиска

Если x — узел бинарного дерева поиска, а узел y находится в левом поддереве x, то

$$key\ [y] \le key\ [x]$$

Если узел y находится в правом поддереве x, то

$$key\ [x] \le key\ [y]$$

Временная сложность большинства операций

$$T(n) = O(h),$$

где h- высота дерева $(h=O(\lg n))$

Бинарные деревья: Поиск

```
TREE_SEARCH(x, k) // x- указатель на
                  // корень, k- ключ
1  if x = nil или k = key[x]
2  then return x
3  if k < key[x]
4  then return TREE_SEARCH(left[x], k)
5  else return TREE_SEARCH(right[x],k)
```

Бинарные деревья: Поиск

```
Iterative_Tree_Search(x, к)
 while x≠nil и k≠ key[x] do
    if k < key[x]
    then x←left [x]
    else  x←right [x]
return x
```

Бинарные деревья: Поиск

```
Tree_Minimum(x)
  while left[x] ≠ nil do
      x ← left[x]
return x
```

```
Tree__Maximum(x)
  while right[x) ≠   nil do
     x ← right[x]
  return x
```

вья: Поиск

p- parent x

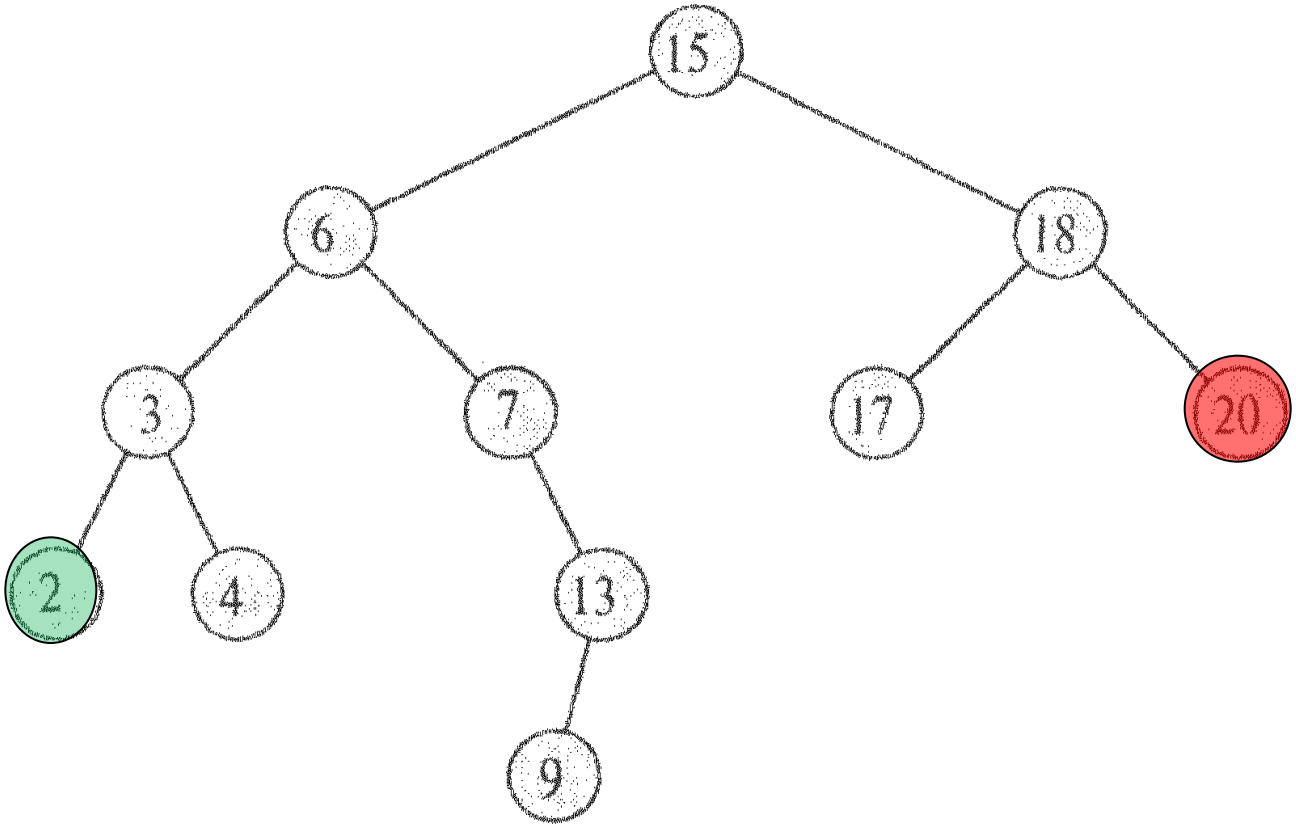
Tree_Su едующий элемент

```
1  if right[x  nil
2   then return TREE_MINIMUM(right[x])
3  y ← p[x]//правое поддерево пустое
4  while y ≠ nil и x = right[y] do
5        x ← y
6        y ← p[y]
7  return y
```

T(n) = O(h)

Бинарные деревья: Вставка

Бинарные деревья



Бинарные деревья: Вставка

- **Для вставки нового значения v в бинарное дерево поиска T - Tree_Insert.**
- **Процедура получает в качестве параметра узел z, у которого key [z] = v, left [z] = nil и right [z] = nil**

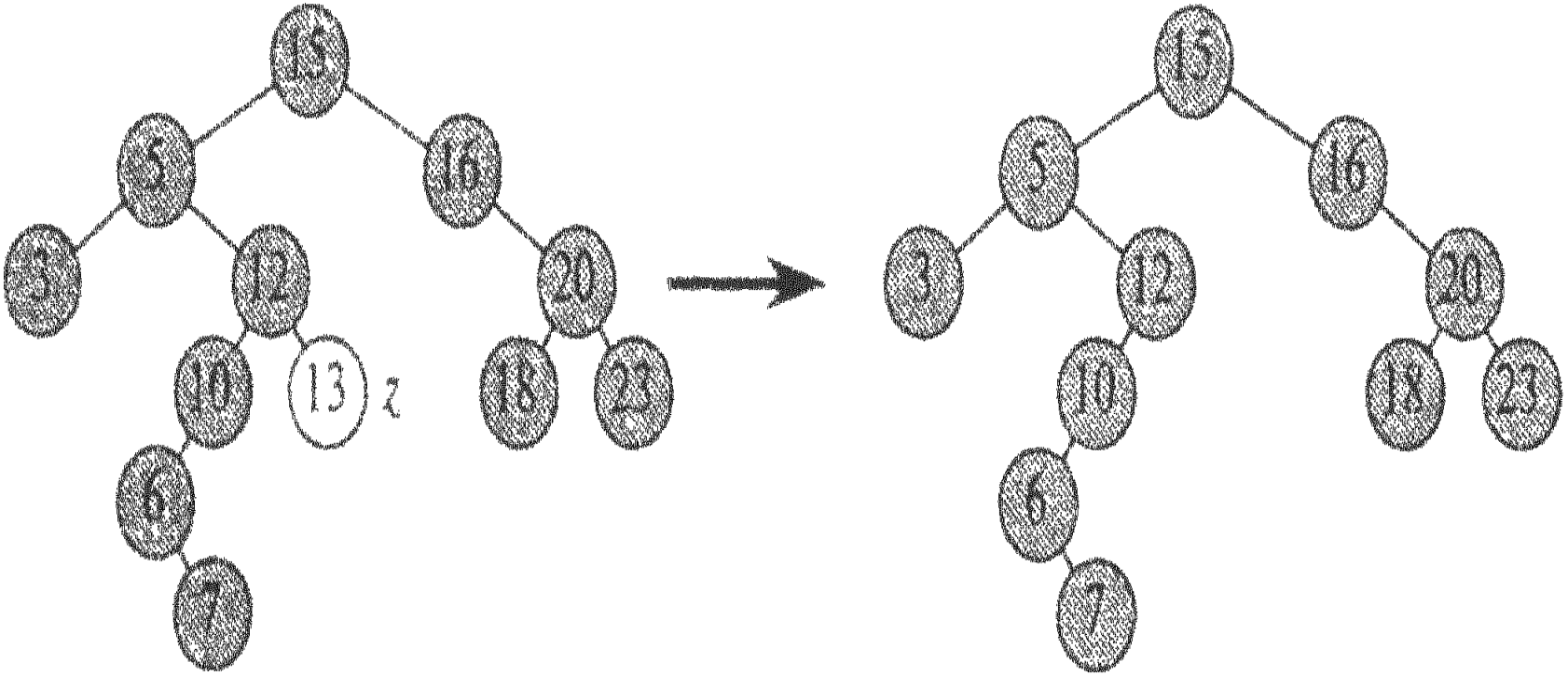
Бинарные деревья: Вставка

```
Tree_Insert(T, z)
1   y ← nil
2   x ← root[T]
3    while x ≠ nil
4      do y ← x //родитель x
5           if key[z] < key[x]
6           then x ← left[x]
7           else x ← right[x]
```

Бинарные деревья: Вставка

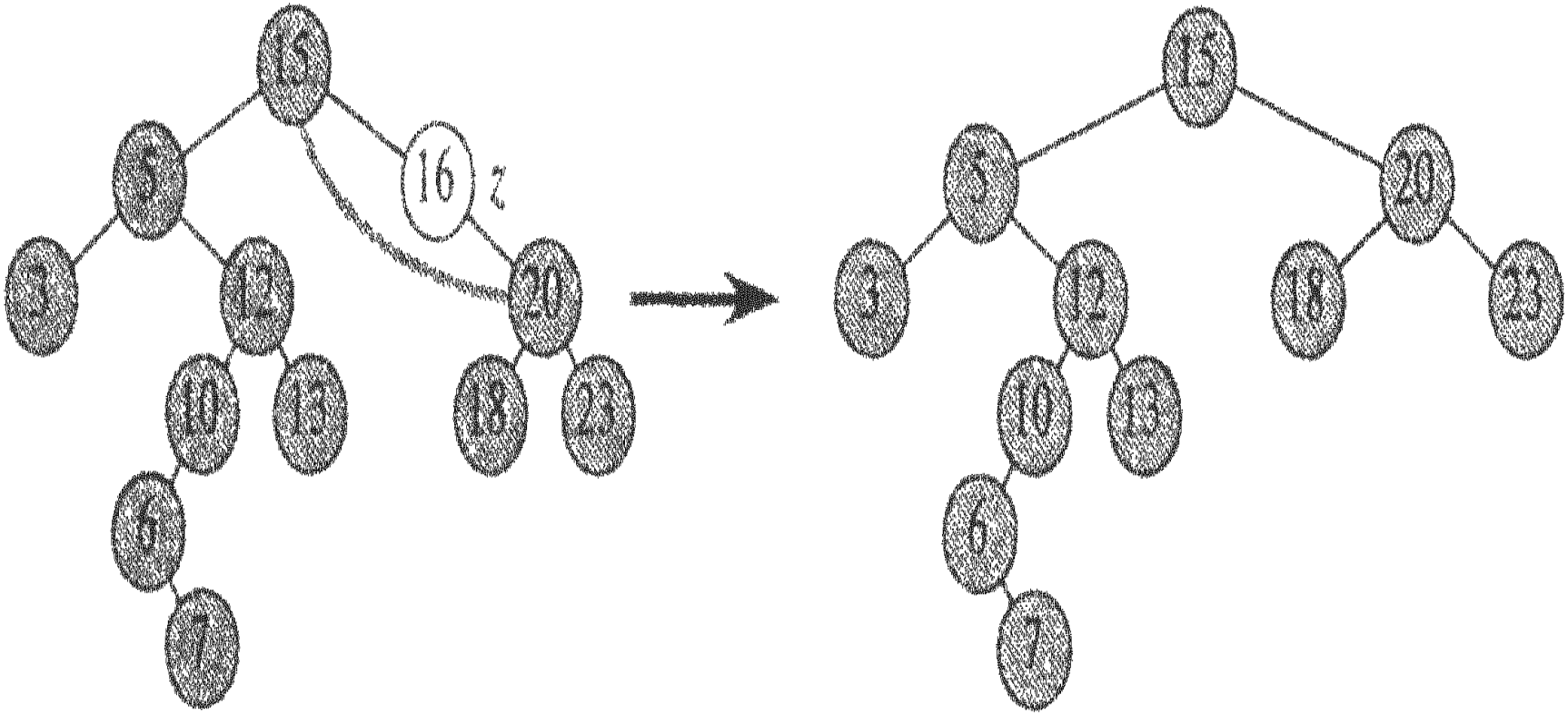
```
8   p[z]  ← y
9    if y = nil
10  then root[T] ← z  //Дерево T — пустое
11  else if key[z] < key[y]
12         then left[y] ← z
13         else right[y] ← z
```

Бинарные деревья: Удаление



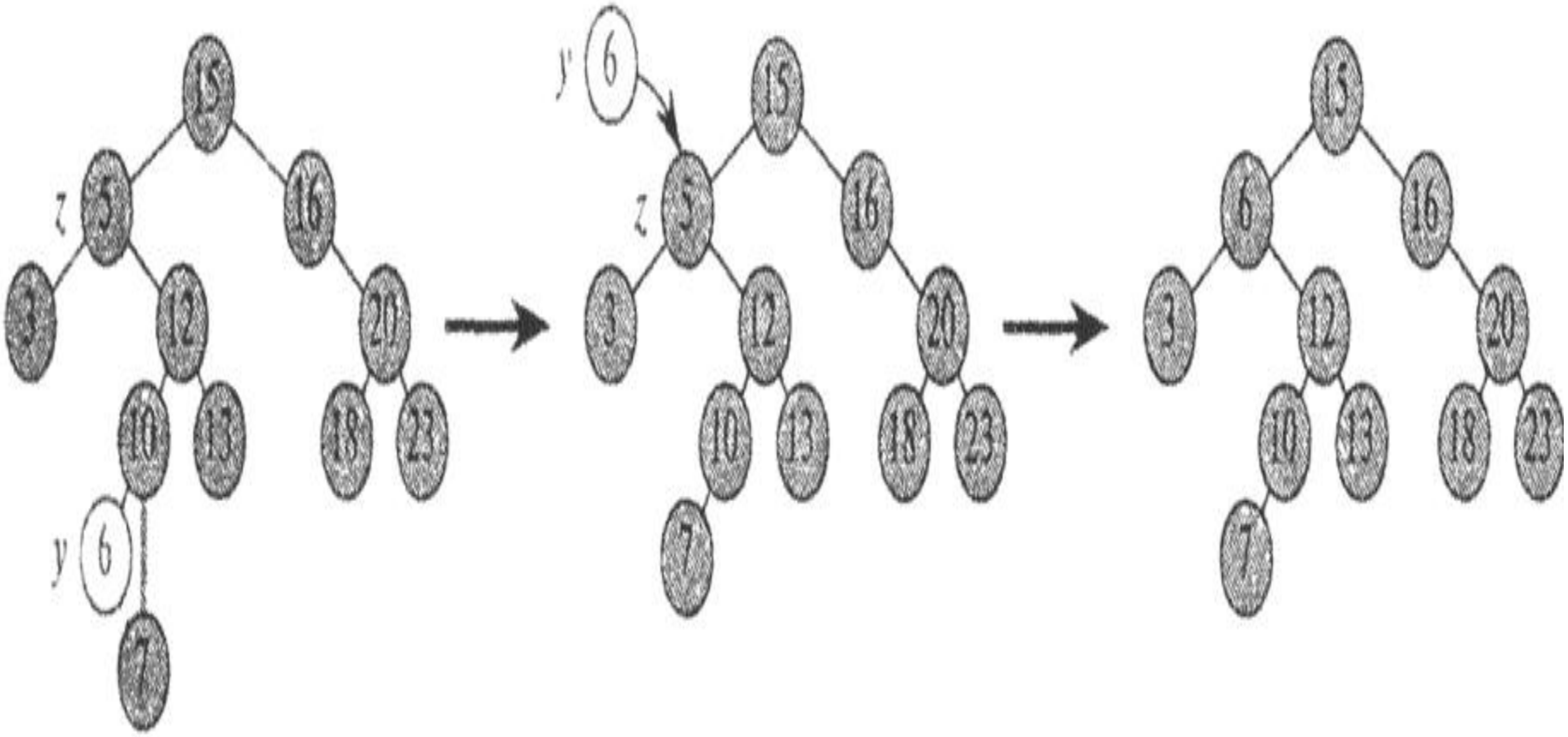
a)

Бинарные деревья: Удаление



б)

Бинарные деревья: Удаление



Бинарные деревья: Удаление

```
Tree_Delete(T, z)
1   if left[z] = nil или right[z] = nil
2     then y ←z //удаляемый узел
3     else y ← Tree_Successor(z)
4   if left[y] ≠ nil
5    then x ← left[y]
6    else x ← right[y]
7   if x  ≠ nil // x – не лист
8     then p[x] ← p[y]
```

Бинарные деревья: Удаление

```
 9   if p[y] = nil
10    then root[T] ← x
11    else if y = left[p[y]]
12           then left[p[y]] ← x;
13           else right[p[y]] ← x
14   if y ≠ z
15    then key[z] ← key[y]
16     // Копирование сопутствующих данных в z
17   return y // для удаления y
```

Бинарные деревья: Удаление

В строках 1-3 алгоритм определяет убираемый путем "склейки" родителя и потомка узел y.

Этот узел представляет собой либо узел z (если у узла z не более одного дочернего узла), либо узел, следующий за узлом z (если у z два дочерних узла).

Бинарные деревья: Удаление

В строках 4-6 x присваивается указатель на дочерний узел узла y либо значение NIL, если у y нет дочерних узлов.

Затем узел y убирается из дерева в строках 7-13 путем изменения указателей в p [y] и x.

Это удаление усложняется необходимостью корректной отработки граничных условий (когда x равно NIL или когда y — корневой узел).

Бинарные деревья: Удаление

Временная сложность алгоритма удаления для дерева высотой h – O(h)

Бинарные деревья: описание класса Tree

```
public class Tree {
/ / Определение класса узла дерева
public static class Node {
  public Object item; // содержимое узла
  public Node left = null; // указатель на левое поддерево
  public Node right = null; //указатель на правое
                                    //поддерево
   / / Конструкторы узла дерева:
  / / конструктор листа
  public Node(Object item) {
     this.item = item;
  }
```

```
 // Конструктор промежуточного узла
public Node (Object item, Node left, Node right) {
  this.item = item;
  this.left = left;
  this.right = right;
 }
}
Node root = null; // корень дерева
}
```

Бинарные деревья

Рекурсивную природу дерева, отраженную в его определении, можно выразить

более явно и в описании класса, если в описании ссылок на поддеревья

вместо класса Node использовать описатель Tree.

Бинарные деревья

Высота бинарного

дерева может быть выражена следующей формулой:

$$h(t) = \begin{cases} 0, & \text{если } t = null; \\ \max(h(t_{left}), h(t_{right})), & \text{если } t \neq null, \end{cases}$$

"правило Леонардо"

"Толщина всех веток дерева на любой его высоте, сложенная вместе, дает толщину ствола".

Бинарные деревья

```
public class Tree {
  private Object item;
  private Tree left = null;
  private Tree right = null;
  public int height () {
    int hi = (left == null ? 0 : left.height ( ) );
    int hr = (right == null ? 0 : right.height());
    return Math.max(hi, hr) + 1 ;
  }
}
```

Бинарные деревья

```
public int height ( )  {
    return height(root);
  }
  private int height(Node n) {
    if (n == null) {
    return 0;
    }
    else {
    return Math.max(height(n.left), height(n.right))  + 1;
  }
}
```

Бинарные деревья (создание/вставка)

Алгоритм:

- **Если дерево пусто, заменить его на дерево с одним корневым узлом ((K,V), nil, nil) и остановиться.**
- **Иначе сравнить K с ключом корневого узла X.**
  - **Если K>=X, рекурсивно добавить (K,V) в правое поддерево T.**
  - **Если K<X, рекурсивно добавить (K,V) в левое поддерево T.**

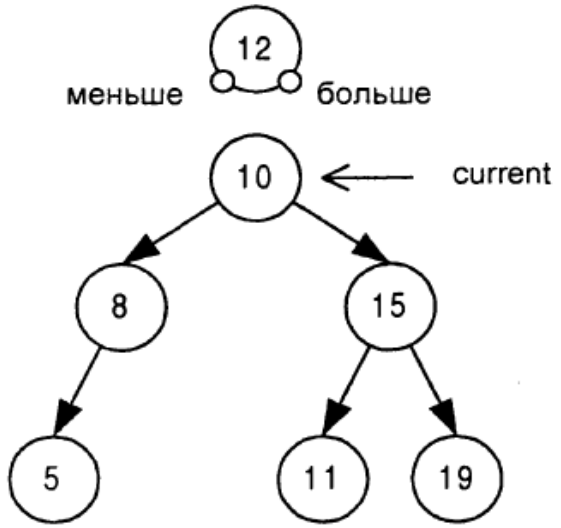
Бинарные деревья (создание/вставка)

```
class Tree {
    public Tree left; // левое и правое поддеревья и ключ
    public Tree right;
    public int key;
    public Tree (int k) { // конструктор с
    //инициализацией ключа
     key = k;
    }
```

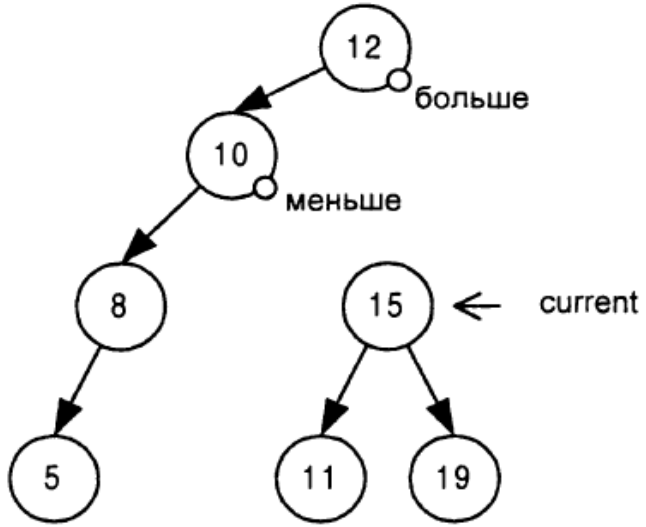
Бинарные деревья (создание/вставка)

```
// insert (добавление нового поддерева (ключа))
    public void insert( Tree  aTree) {
    if ( aTree.key < key )
      if ( left != null ) left.insert( aTree );
      else left = aTree;
    else if ( right != null ) right.insert( aTree );
          else right = aTree;
}
```

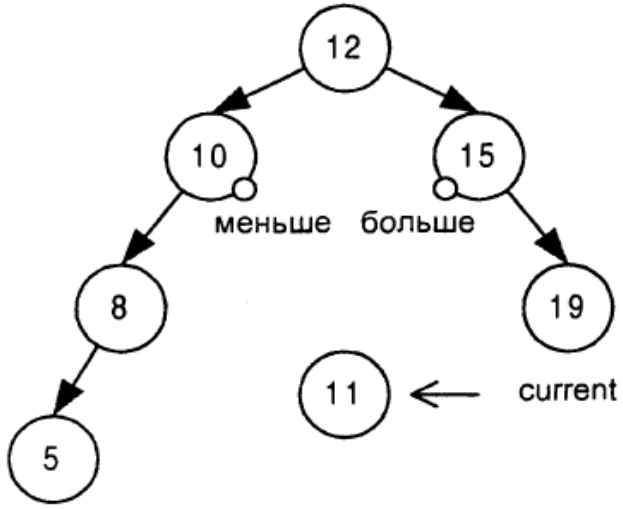
Бинарные деревья (вставка в корень)



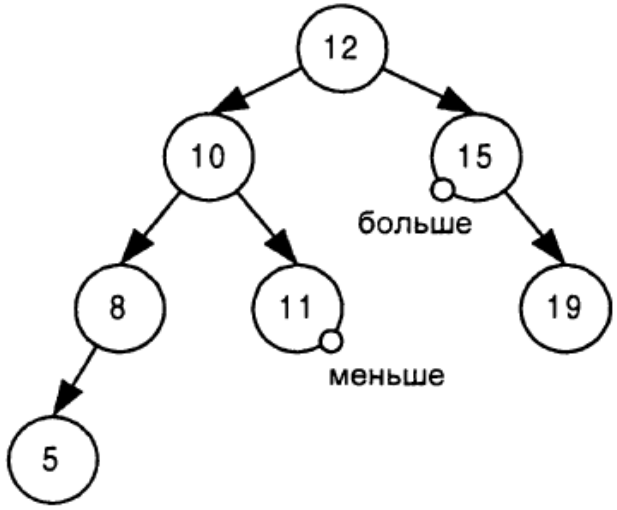
а)



б)



в



г

Бинарные деревья (удаление)

- **Алгоритм:**
- **Если дерево T пусто, остановиться**
- **Иначе сравнить K с ключом X корневого узла n.**
  - **Если K>X, рекурсивно удалить K из правого поддерева T.**
  - **Если K<X, рекурсивно удалить K из левого поддерева T.**

Бинарные деревья (удаление)

Если K=X, то необходимо рассмотреть три случая.

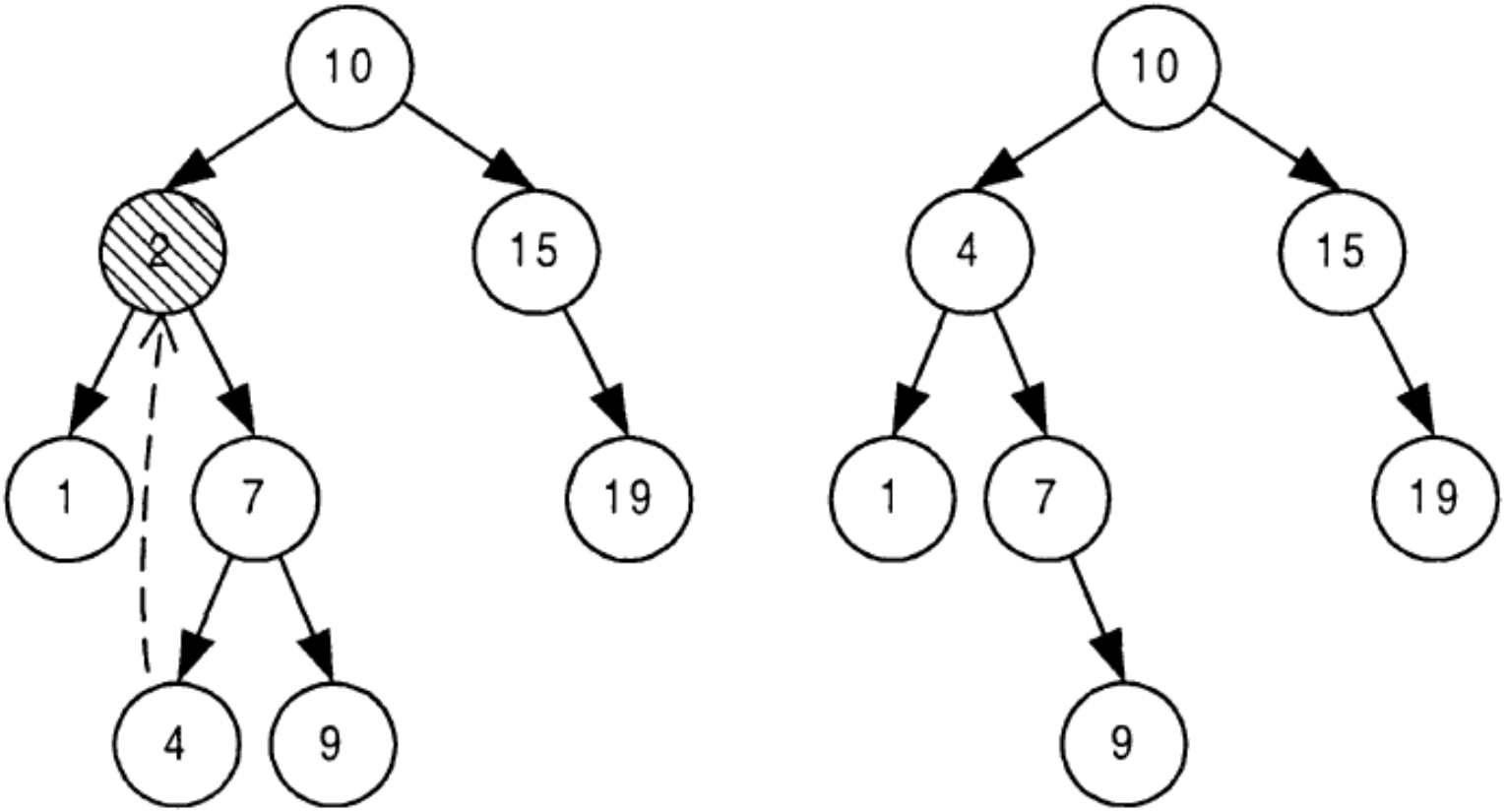
- **Если узел – лист, обнуляем ссылку на него у родительского узла.**
- **Если у узла один потомок, то он и заменяет своего родителя**

Бинарные деревья (удаление)

Если оба поддерева присутствуют, то

- **найдём узел m, являющийся самым левым узлом правого поддерева;**
- **скопируем значения полей (key, value) узла m в соответствующие поля узла n.**
- **у родителя узла m заменим ссылку на узел m ссылкой на правого потомка m (который, в принципе, может быть равен nil).**
- **освободим память, занимаемую узлом m (на него теперь никто не указывает, а его данные были перенесены в узел n).**

Бинарные деревья (удаление)



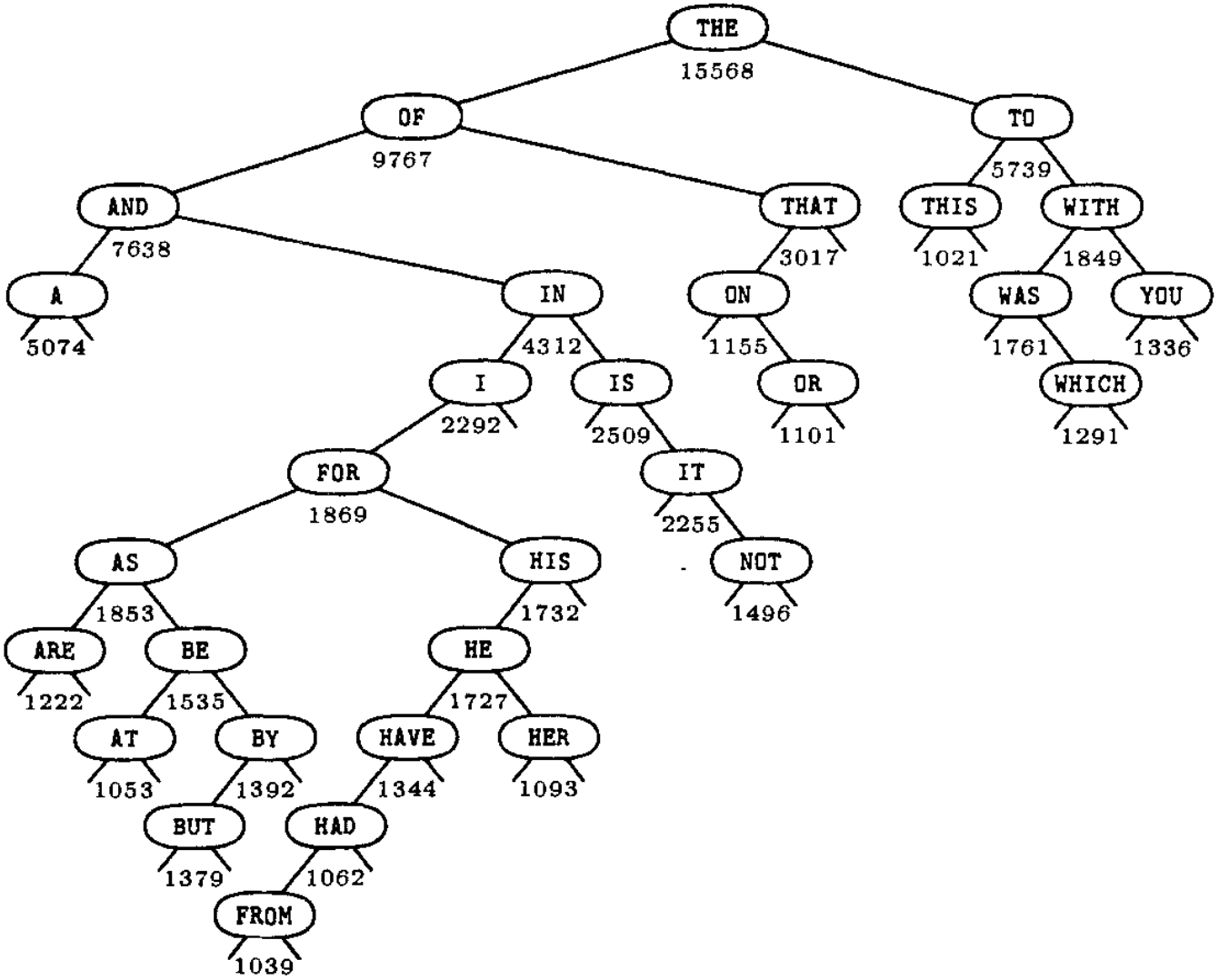
Деревья оптимального поиска

Довольно часто встречаются ситуации, когда есть информация о вероятностях обращения к отдельным ключам.

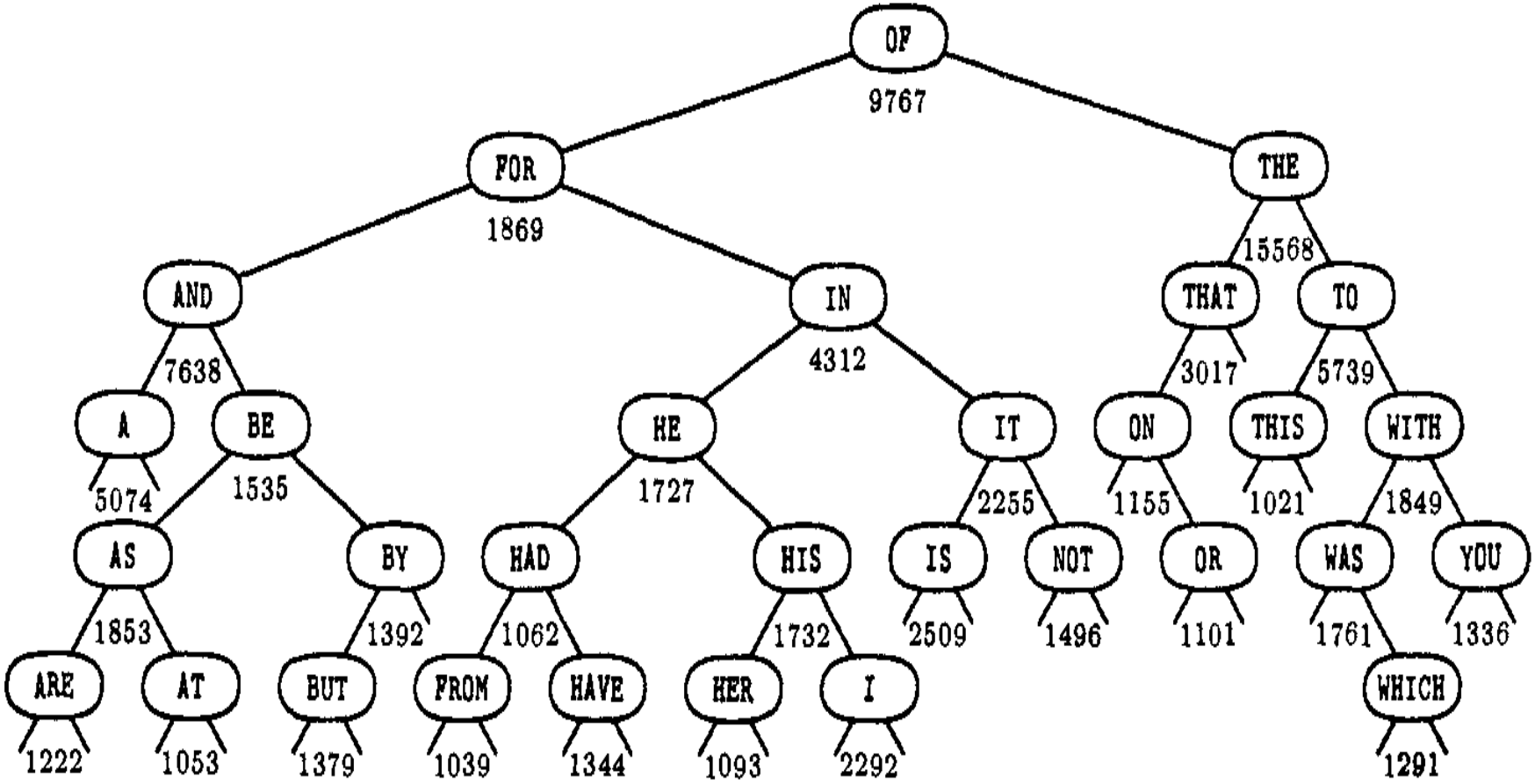
Обычно для таких ситуаций характерно «постоянство» ключей, т.е. в дерево поиска не включаются новые ключи и не исключаются старые, структура дерева остается неизменной.

- **сканер транслятора**
- **электронные словари**
- **системы распознавания текста и тп**

Деревья оптимального поиска



Деревья оптимального поиска



Деревья оптимального поиска

Предположим, что в дереве поиска вероятность обращения к вершине i равна

p_i

$$\Sigma\, p_i = 1,\ \ i=1,\ldots,n$$

Требуется так организовать дерево поиска, чтобы общее число шагов поиска, подсчитанное для достаточно большого числа обращений, было минимальным.

Деревья оптимального поиска

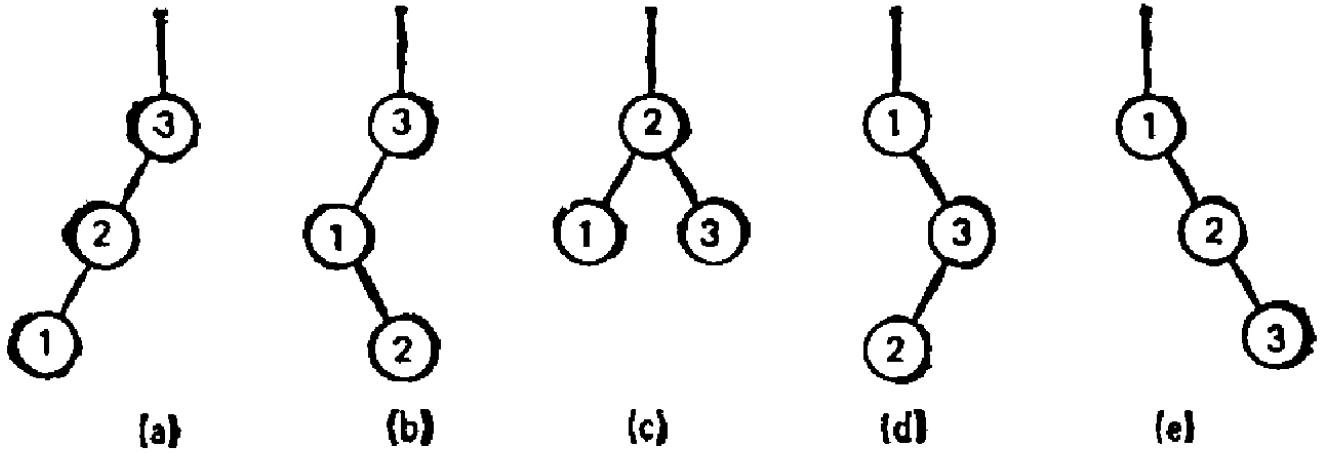
Тогда (внутренняя) *взвешенная длина пути* представляет собой сумму всех путей от корня к каждой из вершин, умноженных на вероятность обращения к этой вершине:

$$p = \Sigma\, p_i * h_i,\ \ i=1,\ldots,n$$

где h_i — уровень вершины i

Конечная цель - минимизировать при заданном распределении вероятностей взвешенную длину пути.

Деревья оптимального поиска



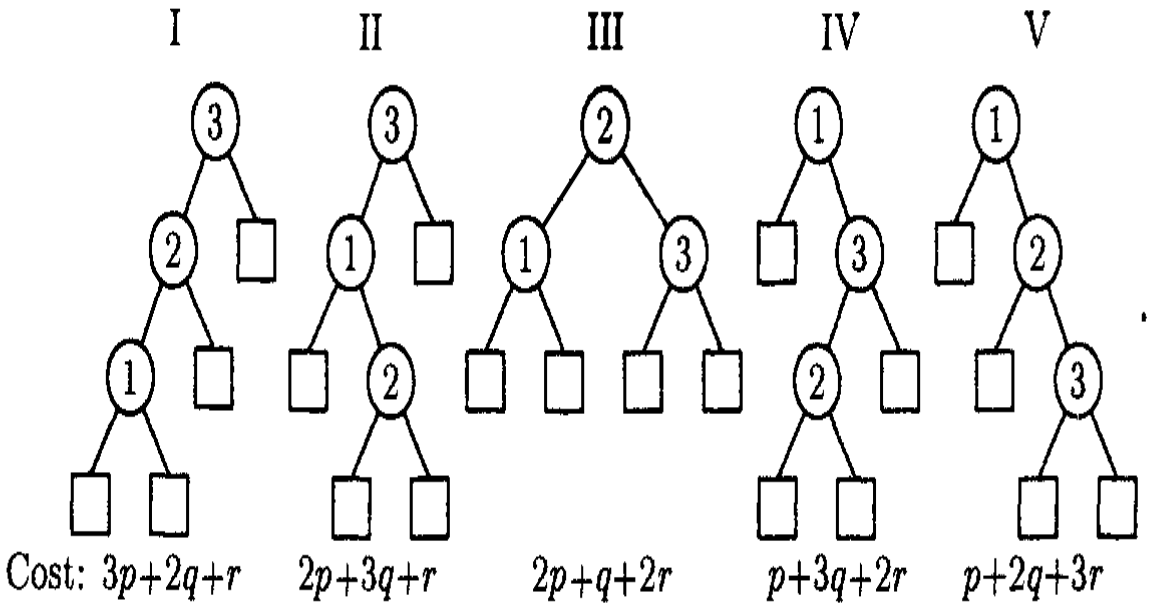
$p_1 = 1/7$, $p_2 = 2/7$ и $p_3 = 4/7$

Взвешенные длины путей деревьев:

P(a)= 11/7, P(b)=12/7, P(c) = 12/7, P(d) = 15/7

P(e) =17/7

Деревья оптимального поиска



Деревья оптимального поиска

Сформулируем задачу следующим образом. Даны 2n + 1 вероятностей $p_1, p_2, \ldots, p_n$ и $q_0, q_1, \ldots, q_n$, где

- **p_i — вероятность того, что аргументом поиска является K_i**
- **q_i — вероятность того, что аргумент поиска лежит между K_i и K_{i+1}.**
- **(q_0 представляет собой вероятность того, что аргумент поиска меньше, чем K_1, а q_n — вероятность того, что аргумент поиска больше, чем K_n.) Таким образом,**
- $p_1 + p_2 \ldots + \ldots p_n + q_0 + q_1 + \ldots + q_n = 1$

Деревья оптимального поиска

$A = \{a_1, a_2, \ldots a_n\}$ – множество элементов, ключи которых упорядочены – $K_1 < K_2 < \ldots < K_n$

$B = \{b_0, b_1, \ldots b_n\}$ – множество элементов, ключи X которых находятся в интервале $K_i < X < K_{i+1}$

для b_0: $X < K_1$, для b_n: $X > K_n$

Дерево бинарного поиска с $n+1$ листьями, представляющими элементы множества B, называется расширенным деревом бинарного поиска

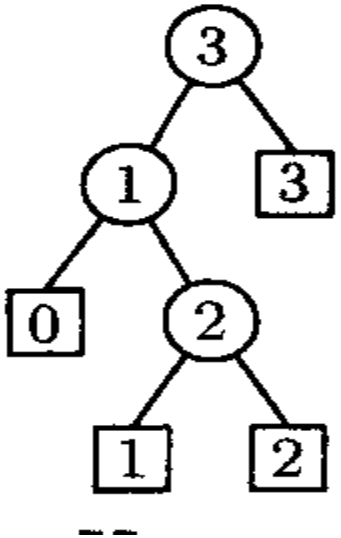
Деревья оптимального поиска

- нужно найти бинарное дерево, минимизирующее ожидаемое количество сравнений при поиске, а именно

$$\sum_{j=1}^{n} p_j \left(\text{уровень}(\textcircled{j}) + 1\right) + \sum_{k=0}^{n} q_k \ \text{уровень}(\boxed{k}),$$

где $\textcircled{j}$ — j-й внутренний узел при симметричном обходе, а $\boxed{k}$ — $(k+1)$-й внешний узел; корень дерева находится на нулевом уровне.

Например, ожидаемое количество сравнений для дерева



равно 2q0+ 2p1+3q1+3p2 +3q2+p3+q3

Деревья оптимального поиска

это значение называется *ценой дерева,*

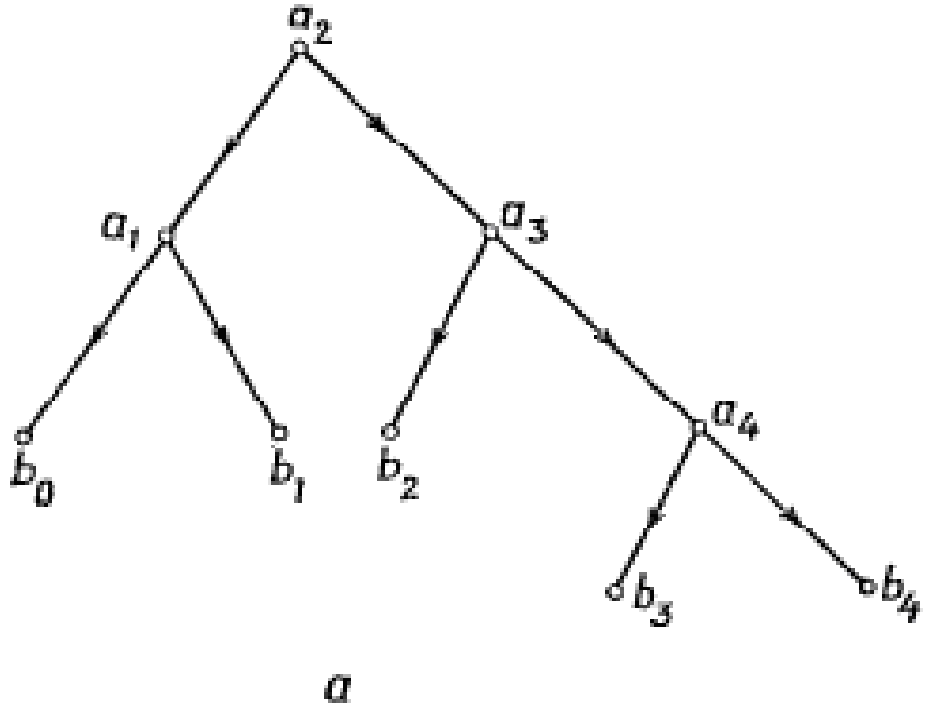
а дерево с минимальной ценой — *оптимальным.* При таком определении требование, чтобы сумма всех *p* и *q* была равна 1, излишне —

мы можем искать дерево с минимальной ценой с любой данной последовательностью весов (p1,..., pn; q0, …, qn)

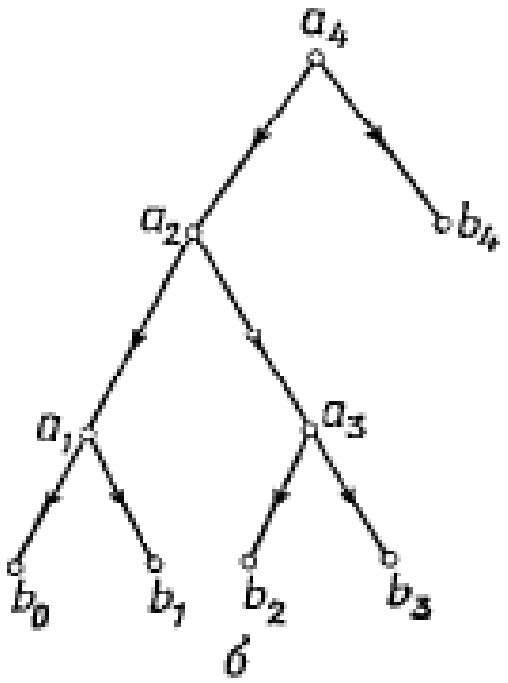
Деревья оптимального поиска

p1=0,2, p2= 0,2, p3= 0,1,p4=0,1

q0=0,1, q1=0,1, q2=0,05, q3=0,05, q4=0,1



а



б

стоимость дерева = 2,05

стоимость дерева = 2,4

Деревья оптимального поиска

Оптимальные деревья обладают одним важным свойством, которое помогает их обнаруживать: все их поддеревья тоже оптимальны;

любое улучшение поддерева должно приводить к улучшению дерева в целом

(схема метода динамического программирования)

Деревья оптимального поиска

- **Пусть *c(i,j)* — цена оптимального поддерева T(i,j) с весами *(pi+1,...,pj, qi,…,qj)***

и пусть

***w(i,j) = p_{i+1} + … + p_j +q_i+ … qj* — сумма этих весов (0<=i<=j<=n)**

Поскольку минимально возможная цена дерева с корнем *(k)* равна

w(i,j) + c(i, k-1) + c(k,j),

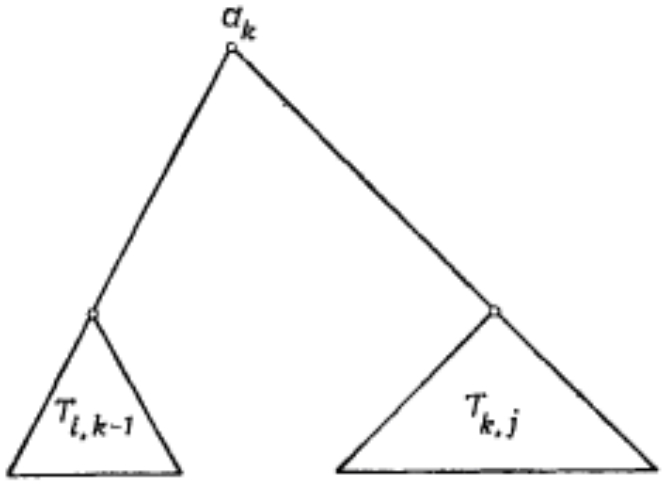
получим

Деревья оптимального поиска

$$c(i,i) = 0,$$
$$c(i,j) = w(i,j) + \min_{i<k\le j}\left(c(i,k-1) + c(k,j)\right) \qquad \text{при } i < j.$$

Через *R(i,j)* при *i < j* обозначим множество всех *k*, для которых достигается этот минимум это множество определяет корни оптимальных поддеревьев.

Деревья оптимального поиска



Пусть T(i,j) для 0<=i<j<=n – оптимальное дерево для {qi,pi+1,qi+1,…pj,qj}

Пусть r(i,j) =ak корень T(i,j)

T(i,i) будет дерево состоящее из листа bi и

c(i,i) = 0, w(i,i) = qi

T(i,k-1) и T(к,j) оптимальное поддеревья

цена: $c(i,j) = w(i,j) + c(i, k-1) + c(k,j),$

Деревья оптимального поиска

- **Алгоритм *поиска оптимальных бинарных деревьев поиска).***

Даны 2n+1 неотрицательных веса (p_1,... ,p_n; q_0,... q_n). Алгоритм строит бинарные деревья *t(i,j)*, имеющие минимальную цену для весов (p_{i+1},.. ..,p_j, q_i,..., q_j). Вычисляются три массива, а именно

- ***c[i,j]*, *0<=i<=j<=n*, цена *t(i,j)*,**
- ***r[i,j]*, *0<=i<j<=n*, корень *t(i,j)*;**
- ***w[i,j]*, *0 <= i <= j <= n*, общий вес *t(i,j)*.**

Деревья оптимального поиска

Результаты работы алгоритма определяются массивом r:

при ***i = j*** ***t(i,j)*** **пуст;**

в противном случае левое поддерево —

t(i, r[i,j]-1),

а правое поддерево — ***t(r[i,j],*** **j).**

Деревья оптимального поиска

- **Алгоритм Гильберта-Мура**

1. **Инициализация:даны p1,p2,..,pn и q0,q1,…qn. Для i=0,1,2,…n положить $w_{ij}=q_i$, $c_{ii}=0$, $r_{ii}=b_i$**
2. **Для l= 1,2,…n выполнить шаг 3**
3. **Для i=0,1,2,…n-l выполнить шаг 4**
4. **Положить j=i+l, $w_{ij}=w_{ij}-1+p_i+q_j$**

Пусть m – значение k (i<k<=j) для которого $c_{ik-1} + c_{kj}$ минимальна.

Положить $c_{ij}=w_{ij} + c_{im-1} +c_{mj}$; $r_{ij}= a_m$

Деревья оптимального поиска

Вычислив r(i,j) строим дерево T(0,n)

1. **r(0,n) - корень дерева, если $r(0,n)=a_k$, то левый сын - r(0,к-1), а правый r(к,n)**
2. **Если $a_m=r_{ij}$ внутренняя вершина, то у a_m левый сын - r(i,m-1), а правый - r(m,j),**

пример

Деревья оптимального поиска

С помощью уравнений можно вычислить $c(i,j)$ для $j - i = 1,2,..., n$;

имеется примерно $1/2n^2$ таких значений, а операция минимизации выполняется примерно для $1/6\ n^3$ значений k.

те можно определить оптимальное дерево за $O(n^3)$ единиц времени с использованием $O(n^2)$ ячеек памяти.

- частоты баланс оптим

АВЛ- деревья

Г.М.Адельсон -Вельский и Е.М.Ландис (1962).

АВЛ-деревья или сбалансированные деревья.

- ***Высотой*** **поддерева будем считать максимальную длину цепи y[1]..y[n] его вершин такую, что y[i+1] – сын y[i] для всех i.**
- **Высота пустого дерева равна 0. Высота дерева из одного корня – единице.**

АВЛ- деревья

- *Идеально сбалансированным* называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более, чем на 1.

АВЛ- деревья

- Дерево считается *сбалансированным по АВЛ* (в дальнейшем просто «сбалансированным»), если для каждой вершины выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1.
- Не всякое сбалансированное дерево идеально сбалансировано, но всякое идеально сбалансированное дерево сбалансировано по АВЛ.

АВЛ- деревья

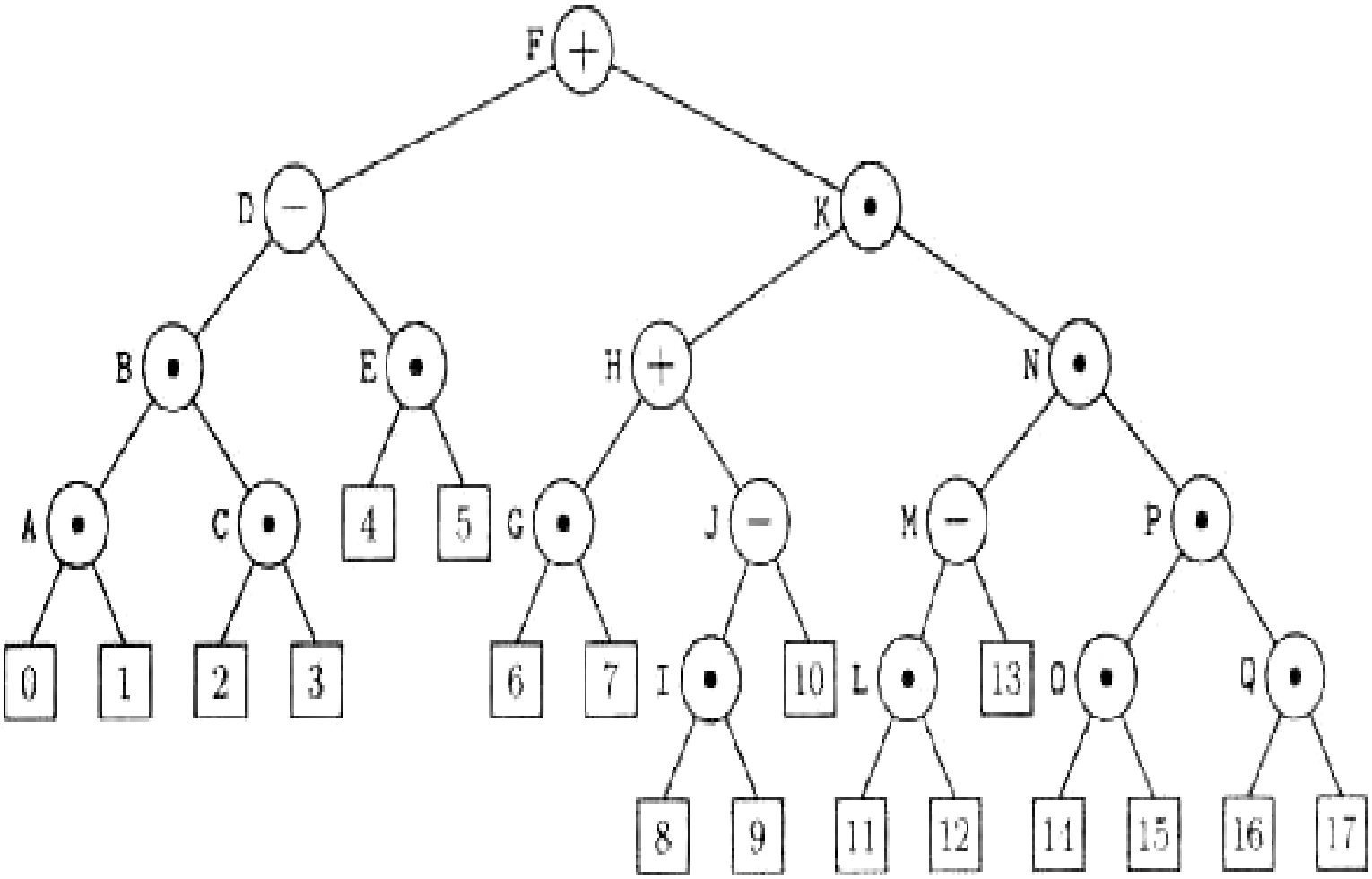
Любая из следующих операций может быть выполнена за t ~ log N:

1. поиск элемента по заданному ключу;
2. поиск k-го элемента по заданному *k;*
3. вставка элемента в определенное место;
4. удаление определенного элемента.

Теорема (Адельсон-Вельский, Ландис)

Высота сбалансированного дерева с N внутренними узлами ограничена значениями

lg (N+1) и 1.4405 lg (N+2) – 0.3277


F
+
D
−
K
•
B
•
E
•
H
+
N
•
A
•
C
•
4
5
G
•
J
−
M
−
P
•
0
1
2
3
6
7
I
•
10
L
•
13
O
•
Q
•
8
9
11
12
11
15
16
17

АВЛ- деревья

```
PTree = ^TTree;
TTree = record
          Item: T; {элемент дерева}
          Left, Right: PTree;
              {указатели на поддеревья}
          Balance: ShortInt;
          {показатель сбалансированности}
        end;
```

АВЛ- деревья

возможный вариант представления на C++ или Java

```
struct node
  { int Key;
    int bal; // Показатель
             //балансированности вершины.
    node *Left;
    node *Right;
    };
```

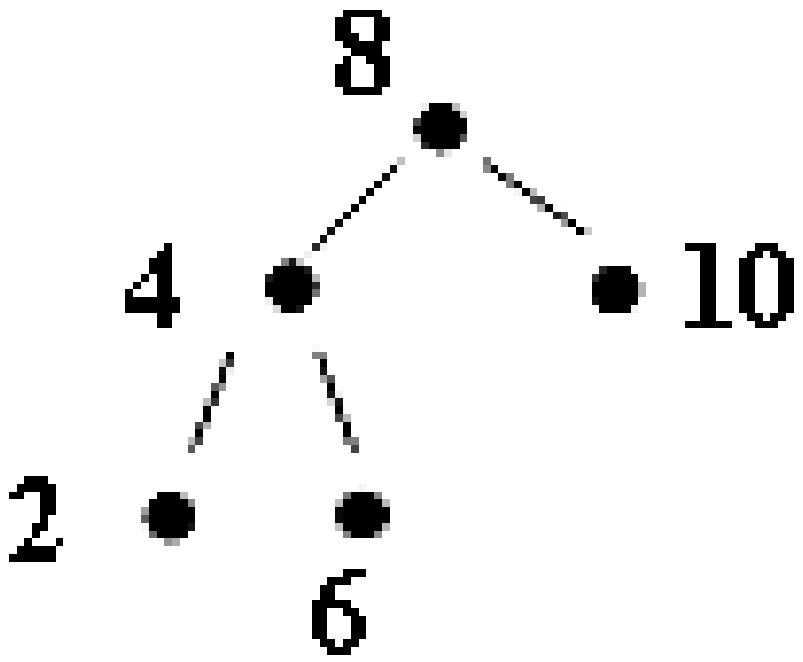
АВЛ- деревья

Включение вершины

1. **hL = hR. После включения L и R станут разной высоты, но критерий сбалансированности не будет нарушен;**
2. **hL < hR. После включения L и R станут равной высоты, то есть критерий сбалансированности даже улучшится;**
3. **hL > hR. После включения критерий сбалансированности нарушится и дерево необходимо перестраивать.**

АВЛ- деревья

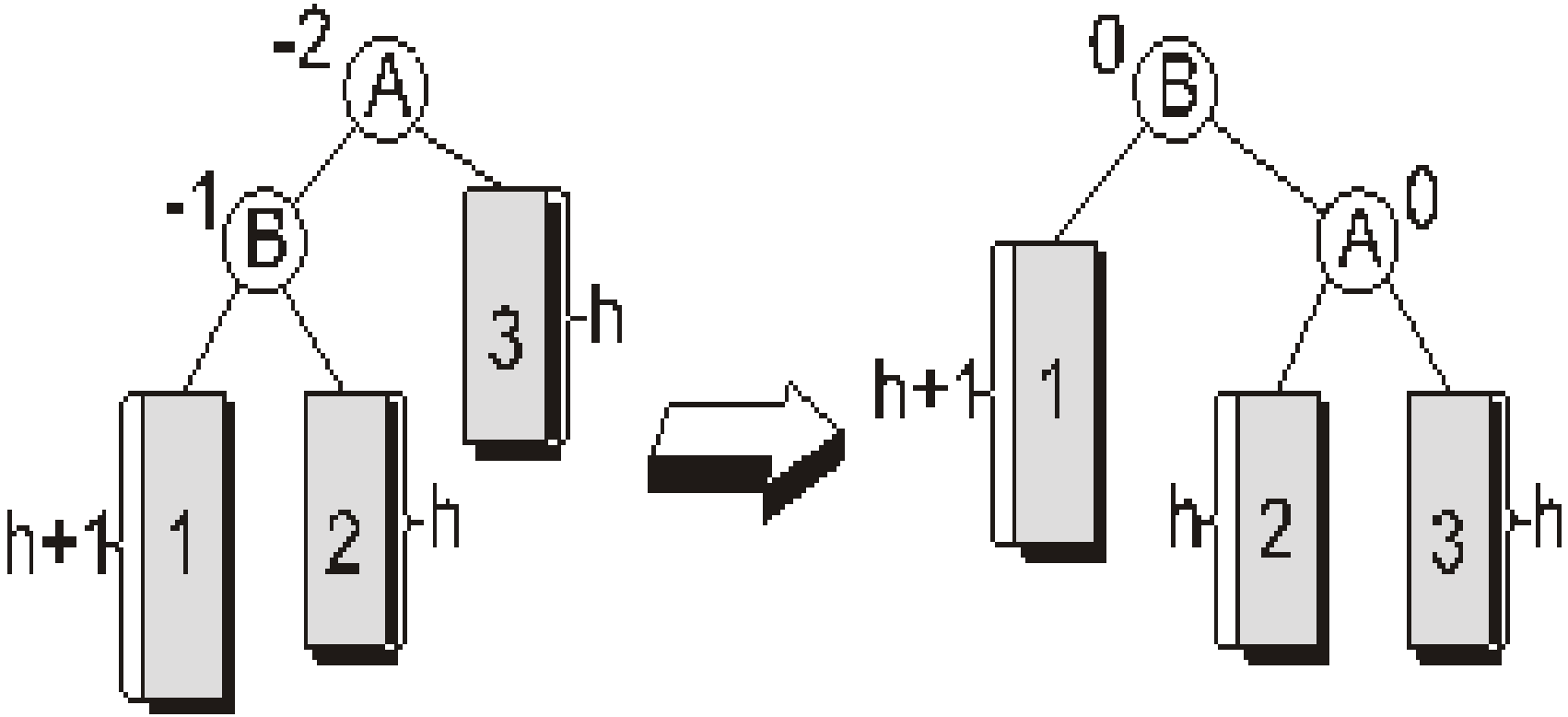
Например:



Алгоритм включения вершины в дерево:

1. **проход по дереву, чтобы убедиться, что включаемого значения в дереве нет;**
2. **включение новой вершины и определение результирующего показателя сбалансированности;**
3. **"отступление" по пути поиска и проверка в каждой вершине показателя сбалансированности. При необходимости - балансировка.**

АВЛ- деревья

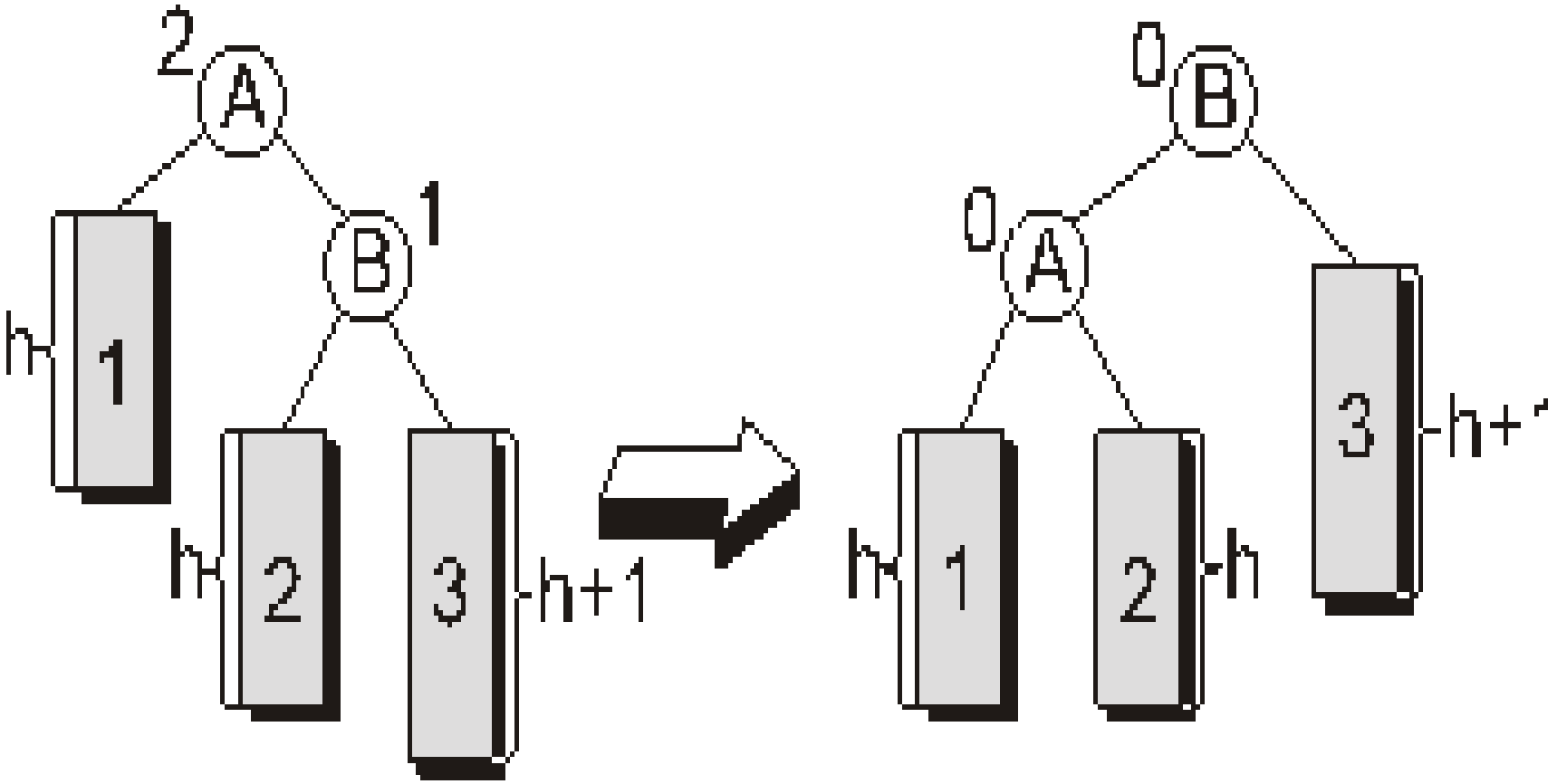


малое левое вращение

АВЛ- деревья

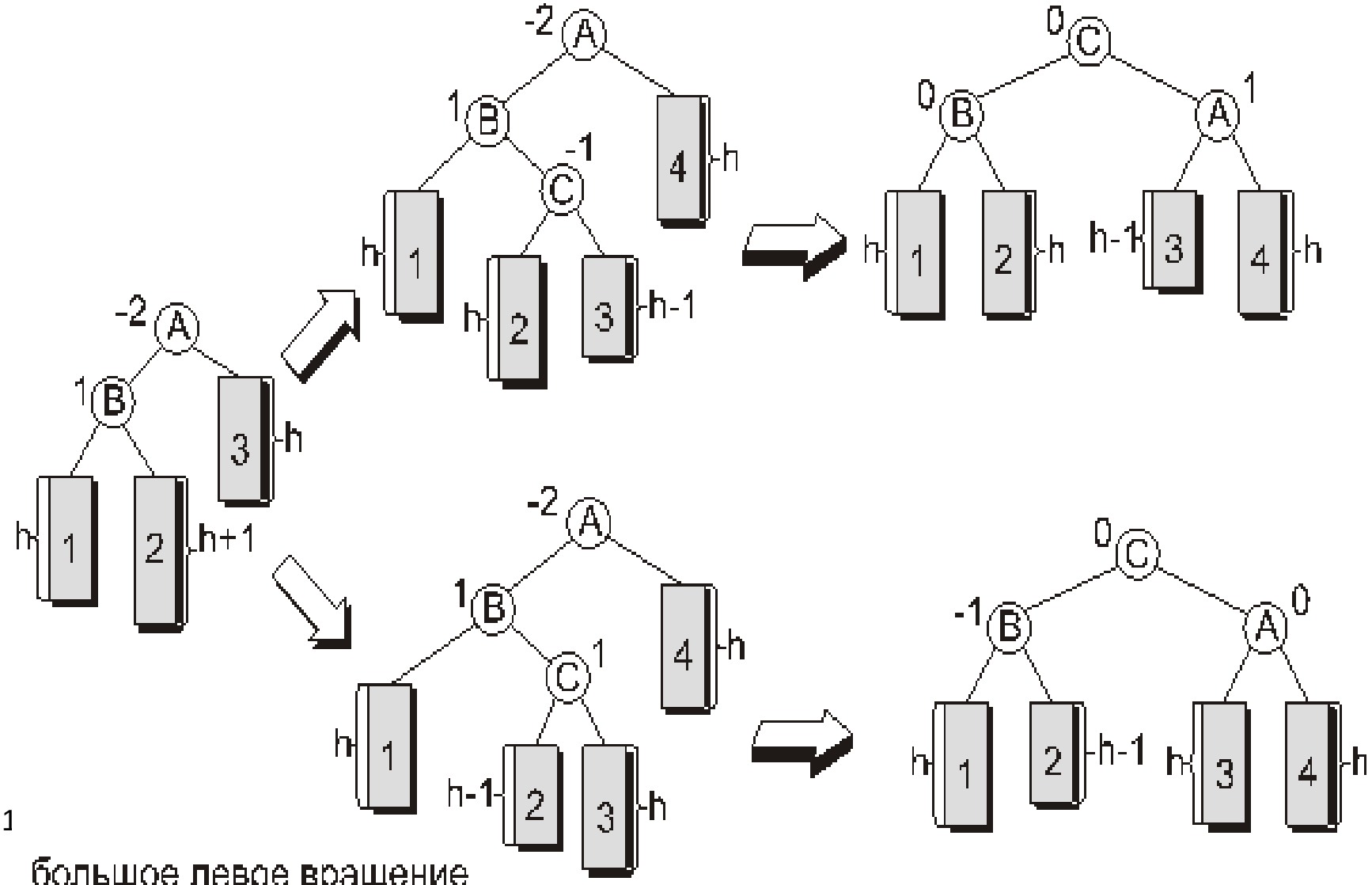
1. **Однократный LL-поворот**
2. **Однократный RR-поворот**
3. **Двухкратный LR-поворот**
4. **Двухкратный RL-поворот**
5. **Построение АВЛ-дерева**

АВЛ- деревья



малое правое вращение

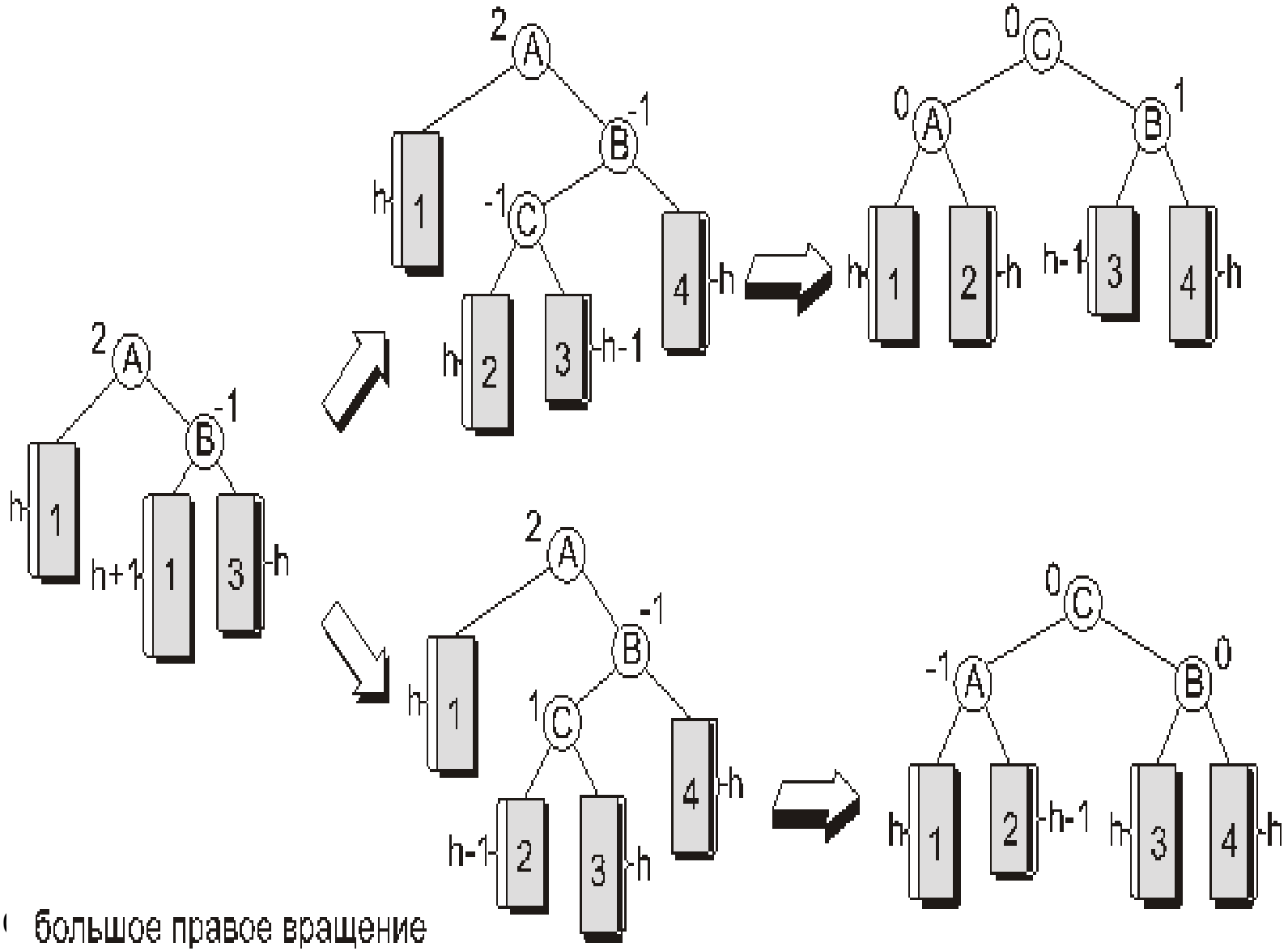
АВЛ- деревья



1

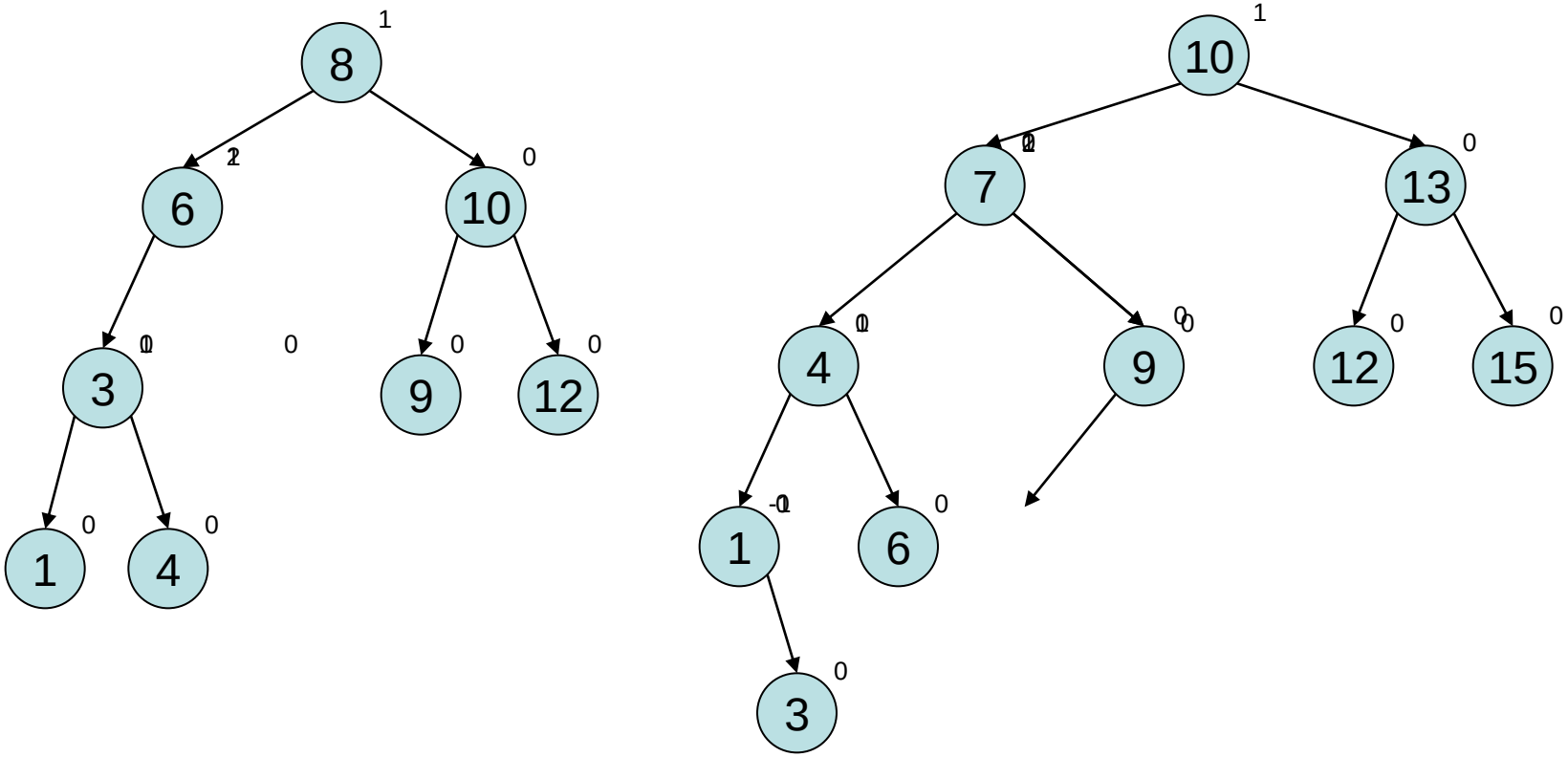
большое левое вращение

АВЛ- деревья



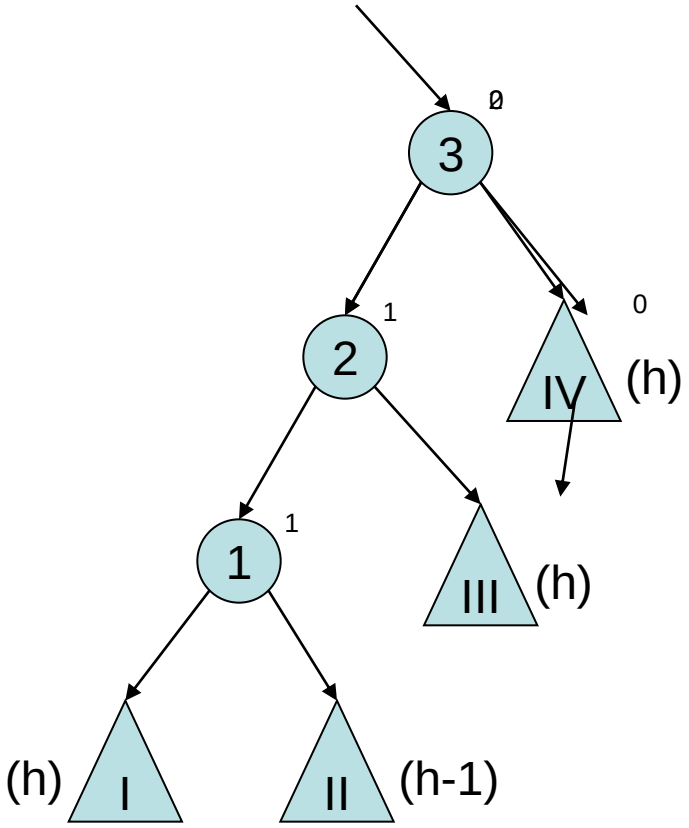
15.(большое правое вращение

Сбалансированные по высоте (АВЛ) деревья



В общем случае трудно придумать алгоритм, позволяющий сбалансировать произвольное дерево, но можно предложить алгоритм балансировки дерева после добавления или удаления одного узла.

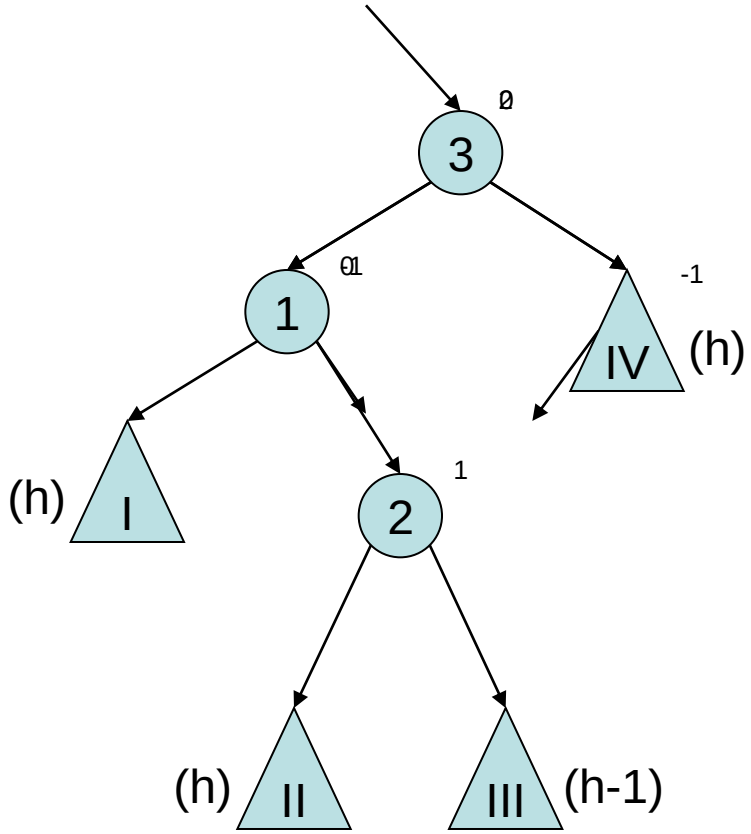
Алгоритм «простого поворота»



1. Отсоединение поддеревьев
2. Поворот
3. Присоединение поддеревьев

(I) < (1) < (II) < (2) < (III) < (3) < (IV)

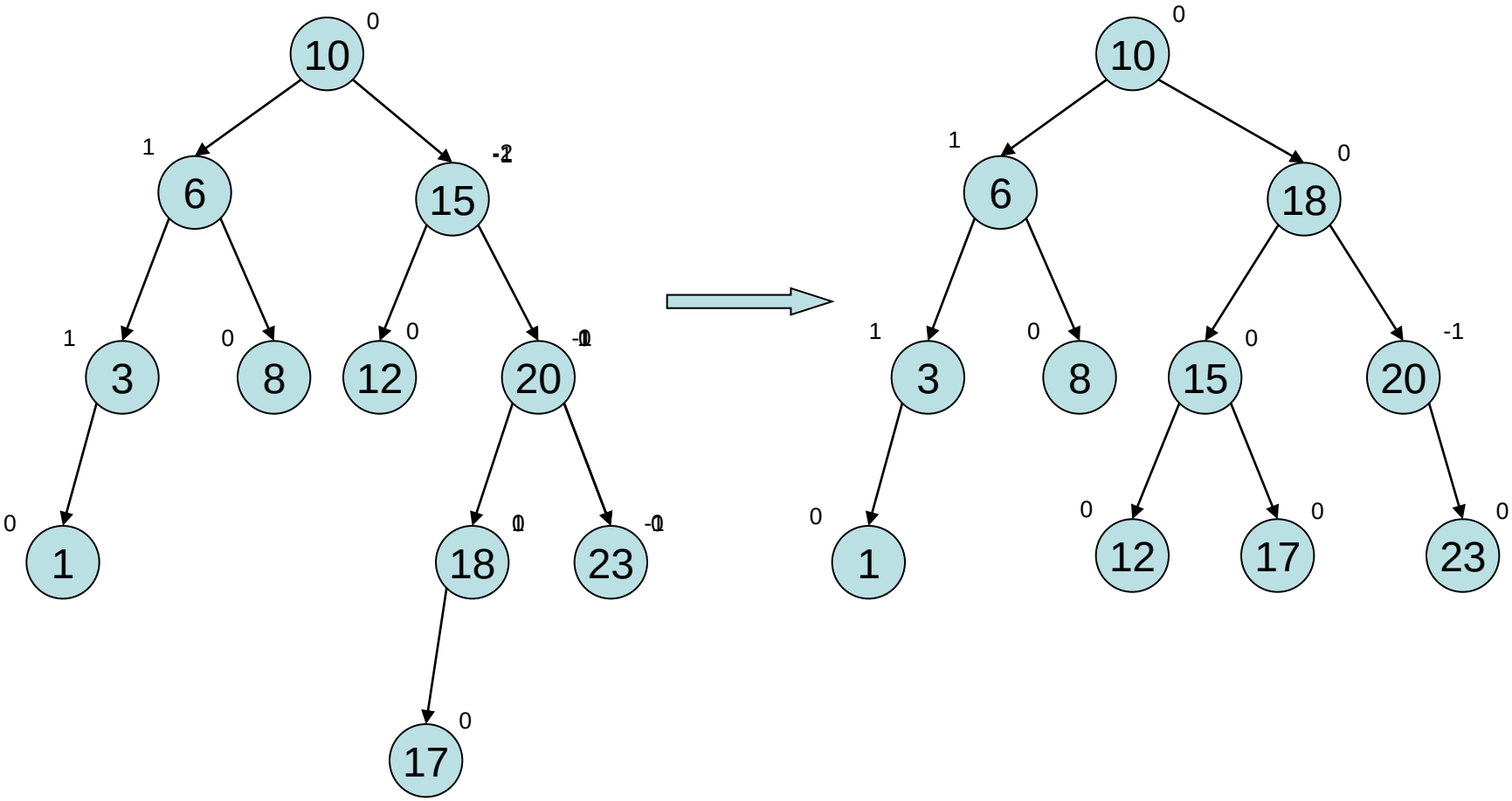
Алгоритм «двойного поворота»



1. Отсоединение поддеревьев
2. Поворот
3. Присоединение поддеревьев

(I) < (1) < (II) < (2) < (III) < (3) < (IV)

Пример вставки ключа



АВЛ- деревья

1. **Математическое ожидание значения высоты при больших n близко к значению $h = \log_2 n + c$, где c - малая константа (c ~0.25).**
2. **Вероятность того, что при вставке не потребуется дополнительная балансировка, потребуется однократный поворот или двукратный поворот, близка к значениям 2/3, 1/6 и 1/6**
3. **Среднее число сравнений при вставке n-го ключа в дерево определяется как $a\log_2 n+b$ (a и b - постоянные).**
4. **Для АВЛ-дерево операции: поиск, вставка и удаление вершины с заданным ключом имеет временную сложность $O(\log_2 n)$.**

2-3-деревья (John Hopcroft, 1970)

определение 2-3-дерева:

- **Каждый узел 2-3-дерева содержит одно или два значения.**
- **Узлы дерева делятся на две категории — листья и промежуточные (внутренние) узлы, причем если промежуточный узел содержит одно значение, то он имеет два непустых поддерева (2-узел), а если он содержит два значения, то он имеет три непустых поддерева (3-узел).**
- **Принцип упорядоченности значений сохраняется для 2-3-дерева . Для 2-узла, как и в случае бинарного дерева поиска, все значения, лежащие в левом поддереве, имеют значения, меньшие значения, хранящегося в узле, а значения, лежащие в правом поддереве, — больше или равны значениям, хранящимся в узле.**

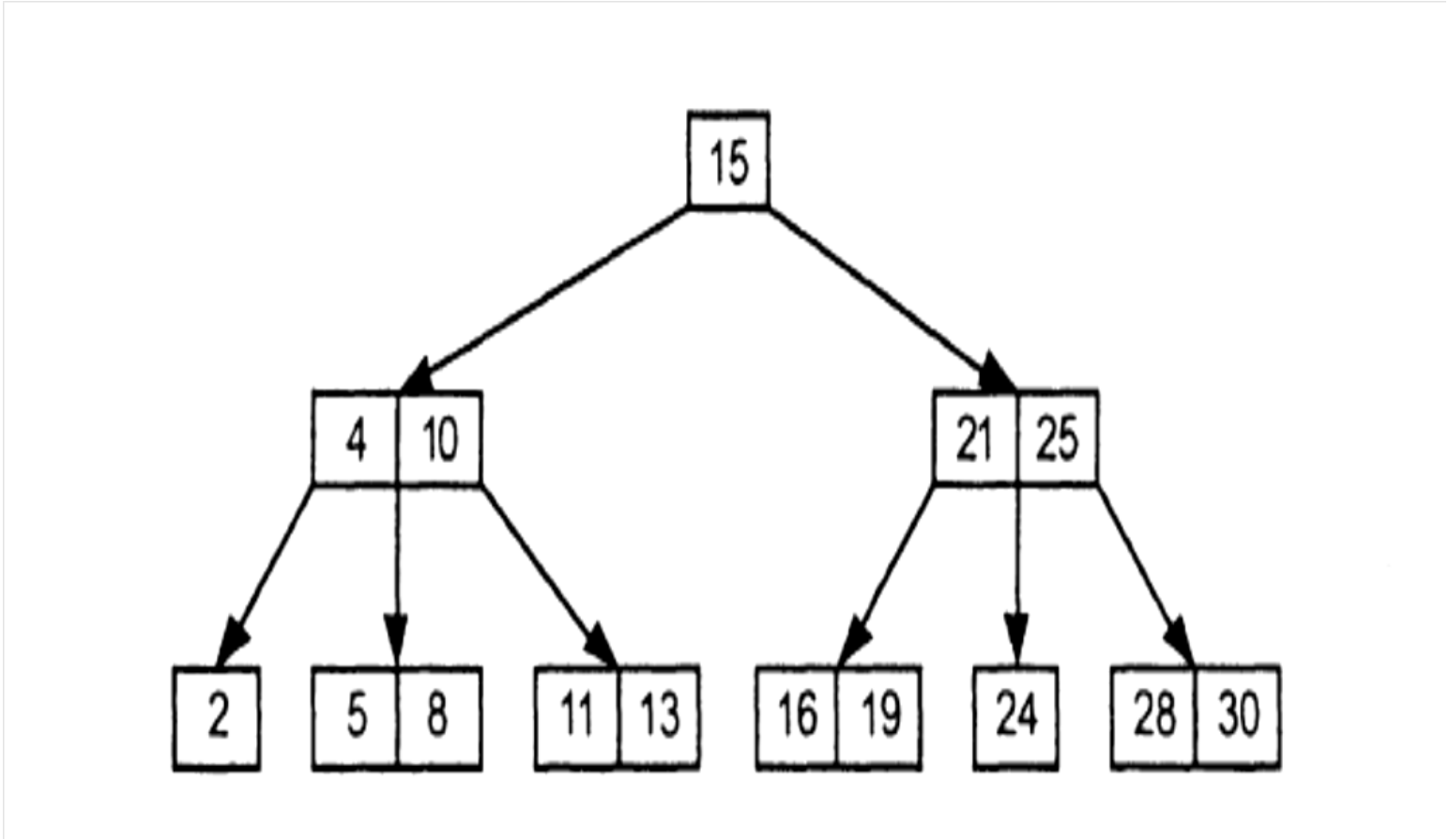
2-3-деревья

Для 3-узла упорядоченность означает следующее: если ключи, хранящиеся в нем, имеют значения K_1, K_2, а поддеревья этого узла обозначены T_1, T_2 и T_3, то справедливо неравенство

$$T_1 < K_1 < T_2 < K_2 < T_3$$

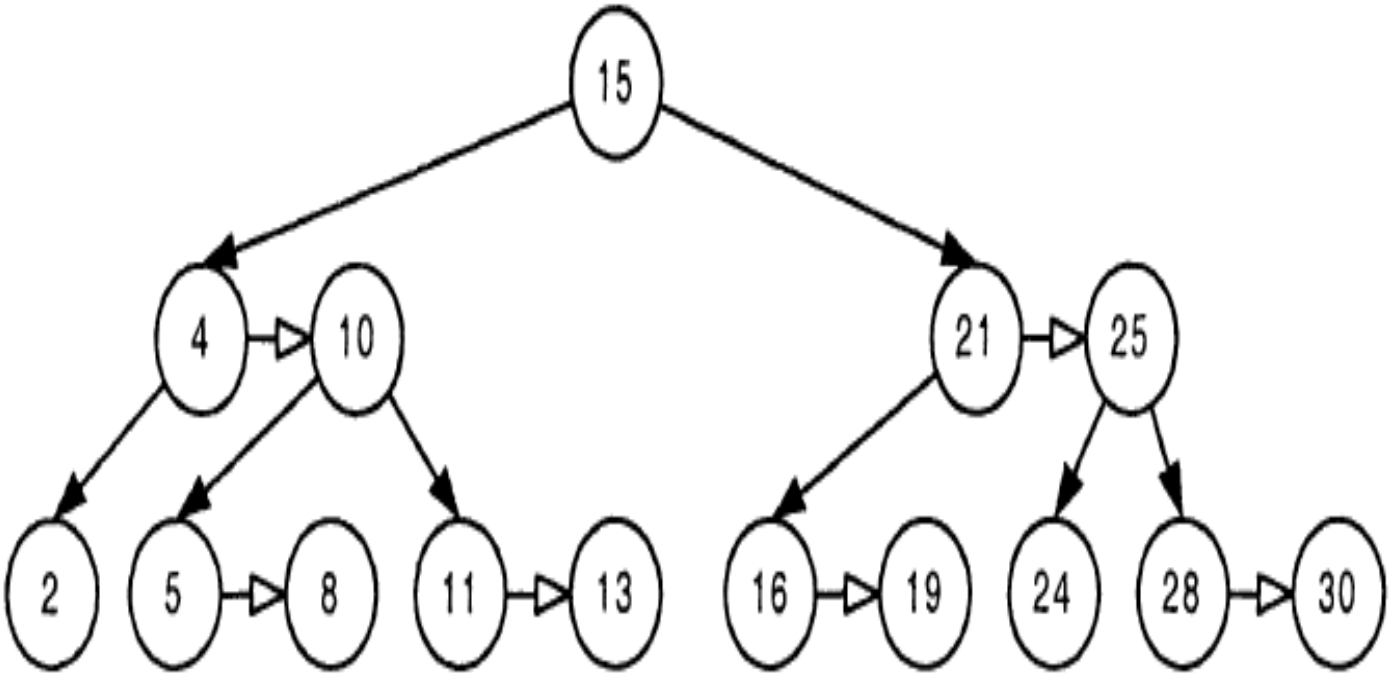
- **Все листья лежат на одном и том же уровне.**

2-3-деревья



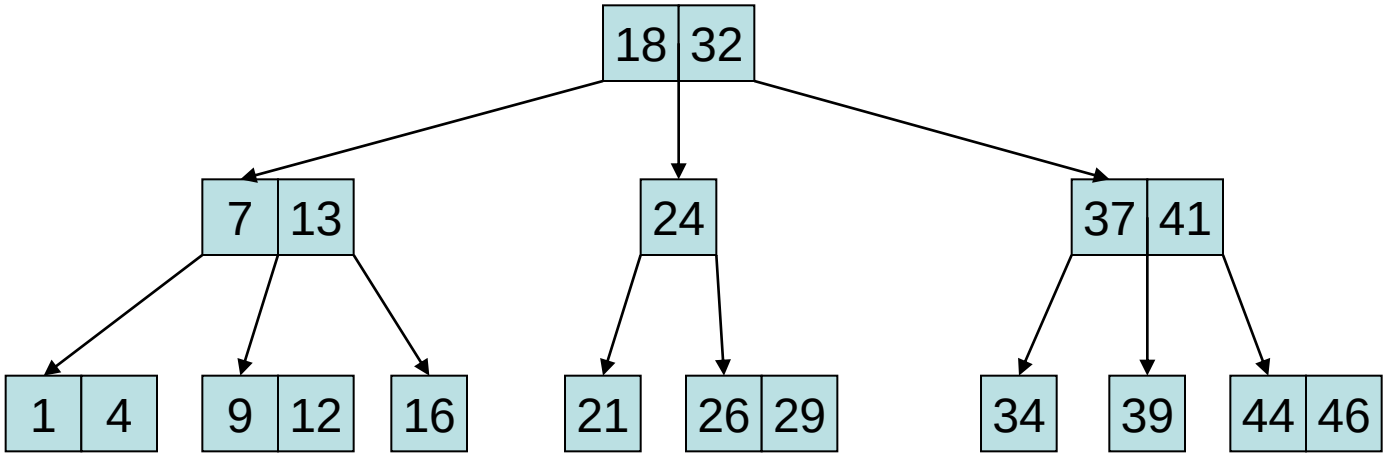
2-3-деревья

представление в виде бинарного дерева



Вставка и удаление

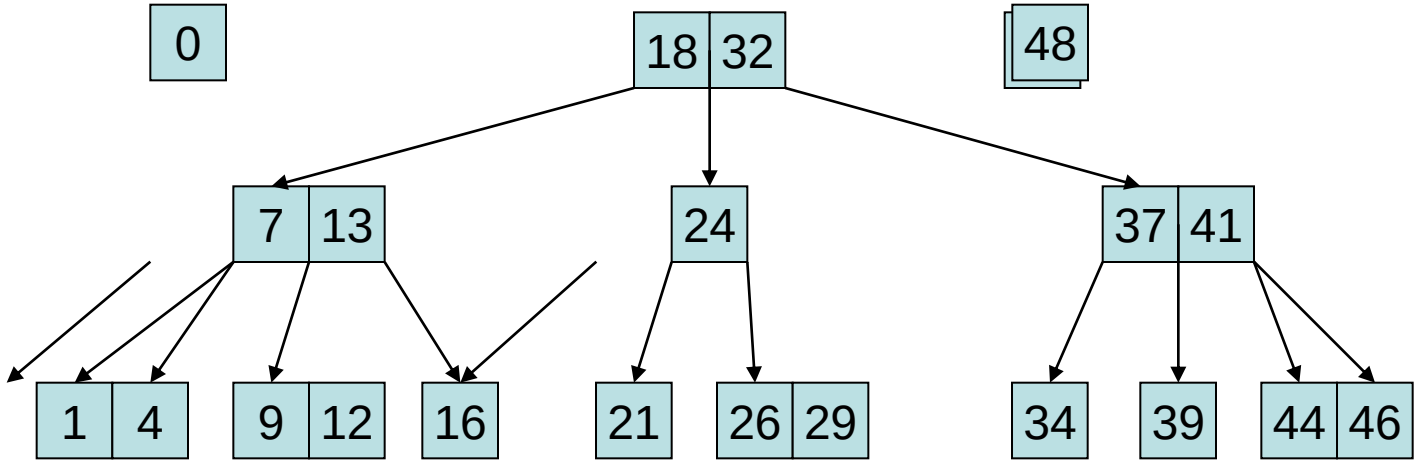
2-3-дерево



Структура 2-3-дерева:

- Каждый узел содержит 1 или 2 ключа
- Ключи упорядочены (возможен быстрый поиск)
- Промежуточные узлы имеют ***все*** ссылки (2 или 3 ссылки)
- Все терминальные узлы (листья) находятся на одном уровне

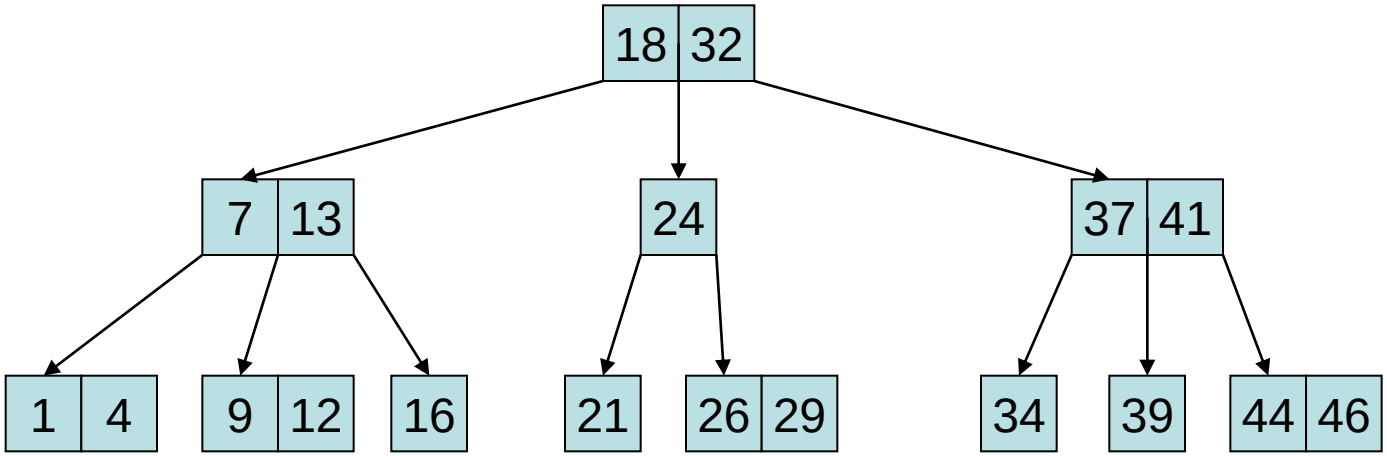
Алгоритмы вставки ключа в 2-3-дерево



Новый ключ вставляется в лист одним из трех методов:

- Расширением терминального узла
- «Переливанием»
- Расщеплением узлов

Алгоритмы удаления ключа из 2-3-дерева



Существующий ключ переносится в лист и удаляется одним из трех методов:

- Сужением терминального узла
- «Переливанием»
- Склеиванием узлов

Красно-чёрные деревья (Red-Black Tree, RB-Tree)

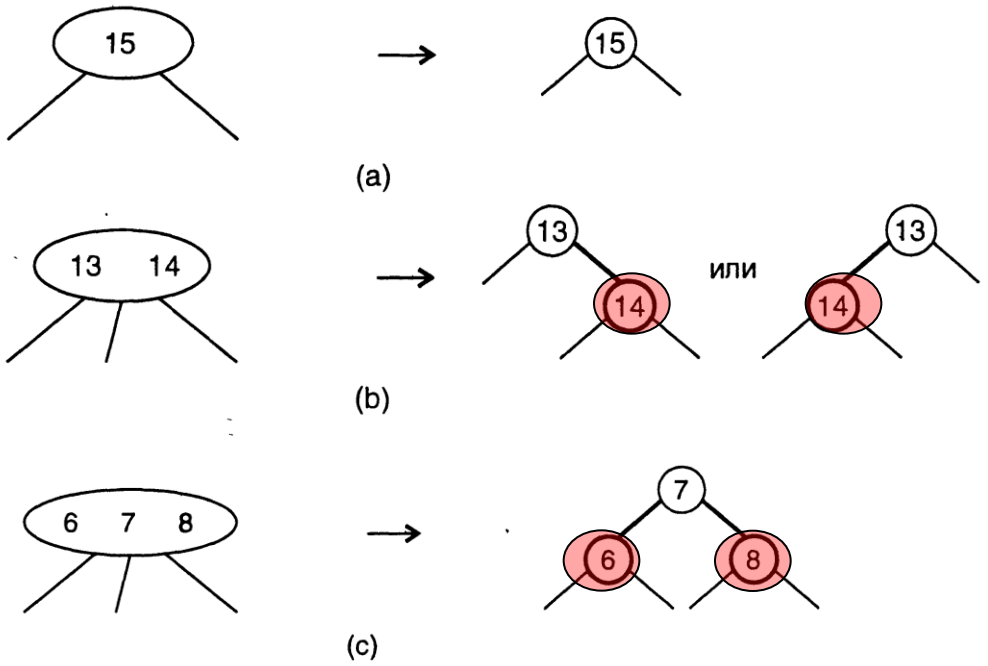
Свойства КЧД (Р. Байер, 1972 г)

- Корень черный;
- Каждая вершина может быть либо красной, либо чёрной. Бесцветных вершин, или вершин другого цвета быть не может.
- Каждый лист (NIL) имеет чёрный цвет.
- Если вершина красная, то оба её потомка – чёрные.
- Все пути от корня к листьям содержат одинаковое число чёрных вершин.

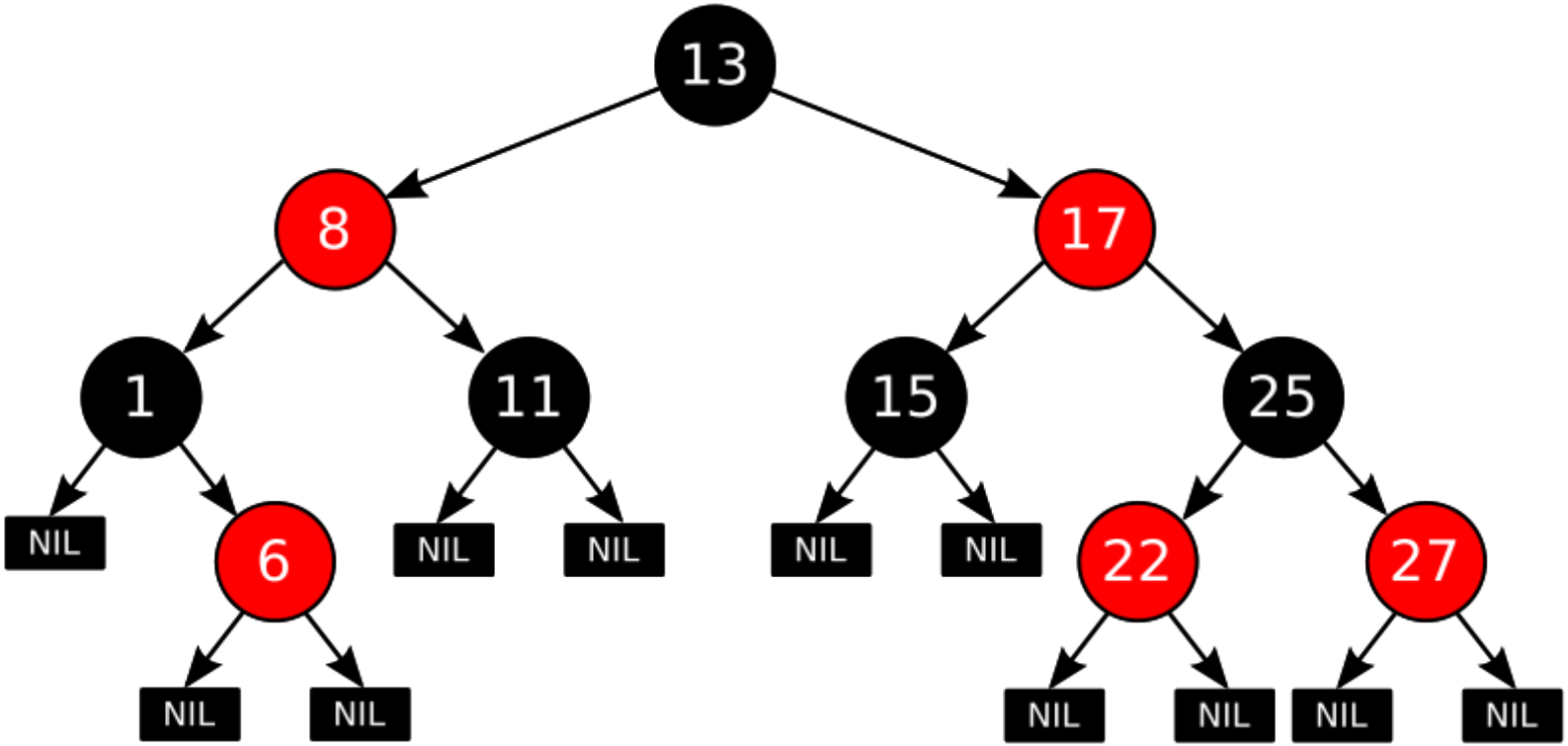
Ни один путь от узла к корню не превышает другого пути более чем вдвое.

Красно-чёрные деревья (Red-Black Tree, RB-Tree)

соответствие 2-3 и 2-4 дерева и КЧД



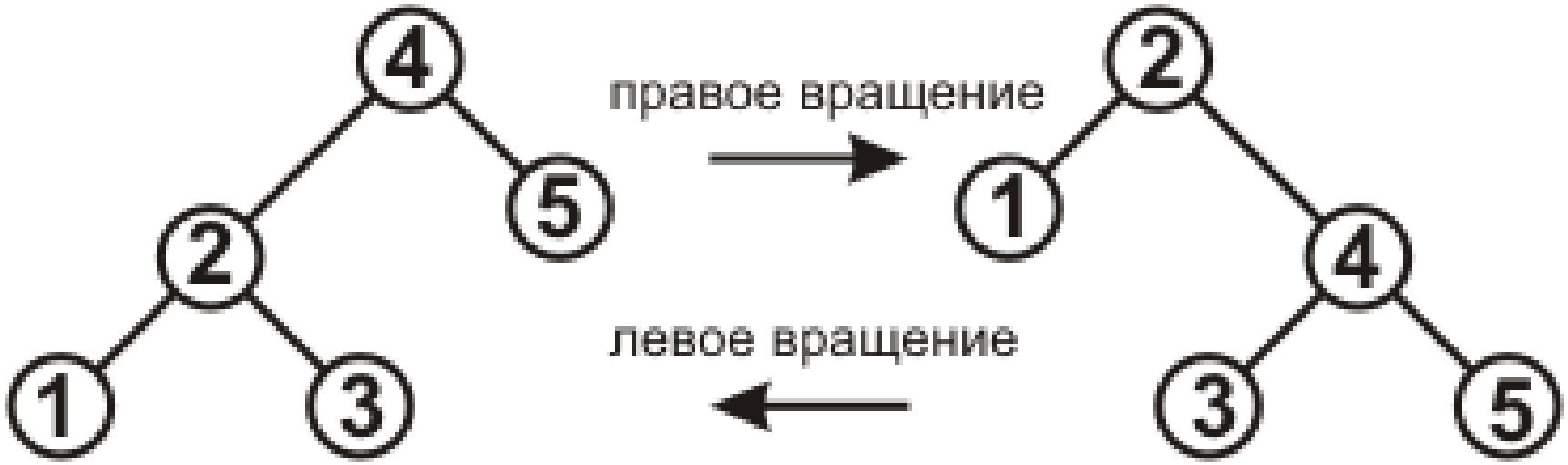
Красно-чёрные деревья (Red-Black Tree, RB-Tree)



Красно-чёрные деревья (Red-Black Tree, RB-Tree)

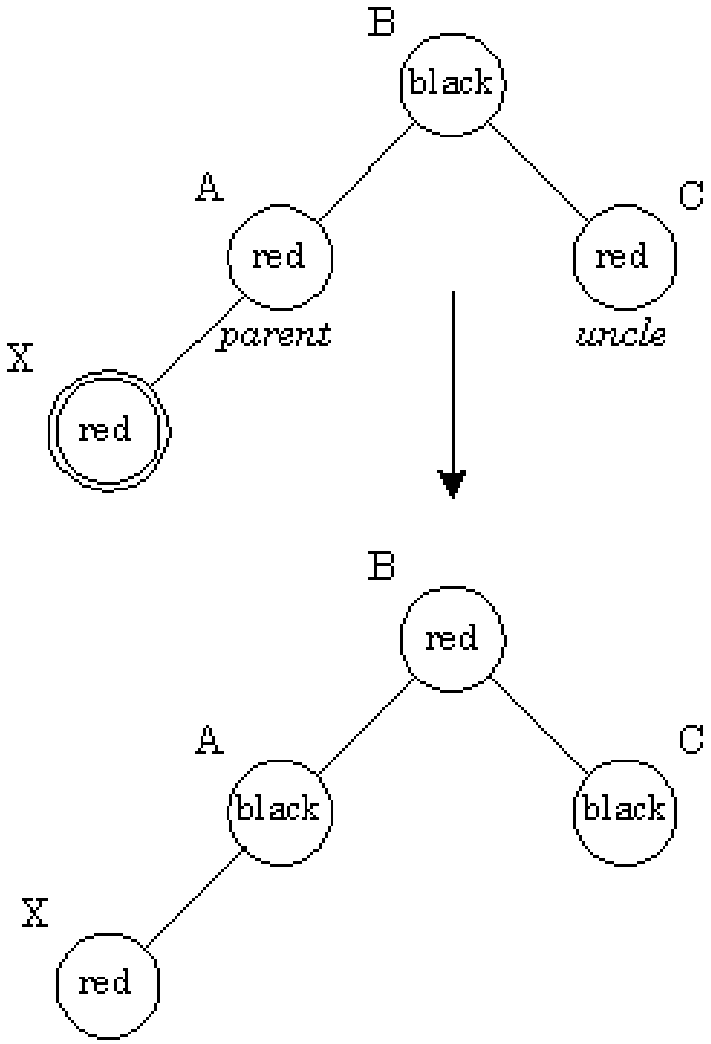
Операции вставки и удаления вершин в КЧД могут нарушать свойства КЧД.

Чтобы восстановить эти свойства, надо будет перекрашивать некоторые вершины и менять структуру дерева



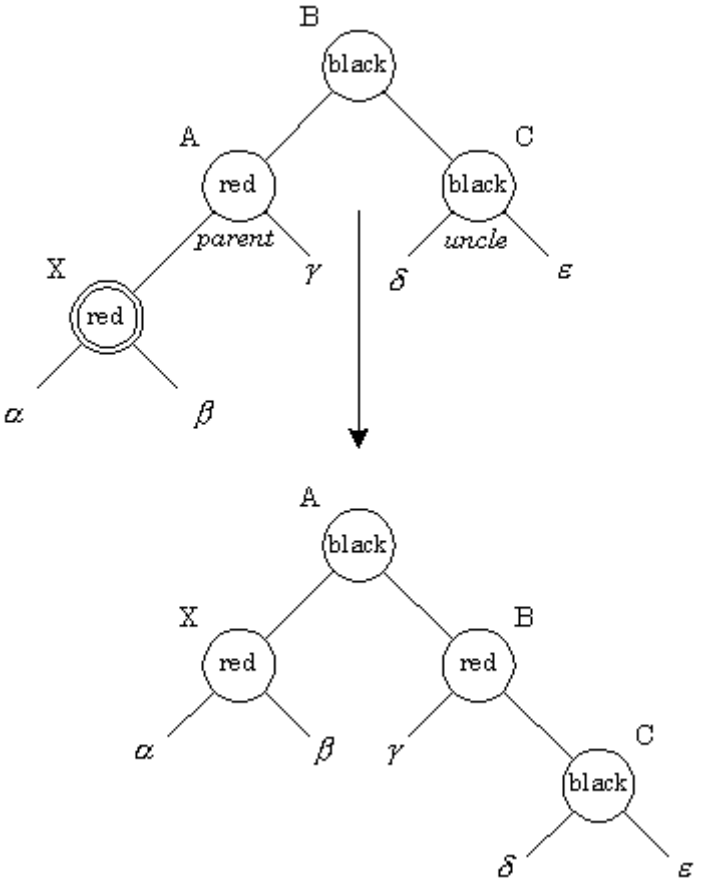
Красно-чёрные деревья (Red-Black Tree, RB-Tree)

Вставка

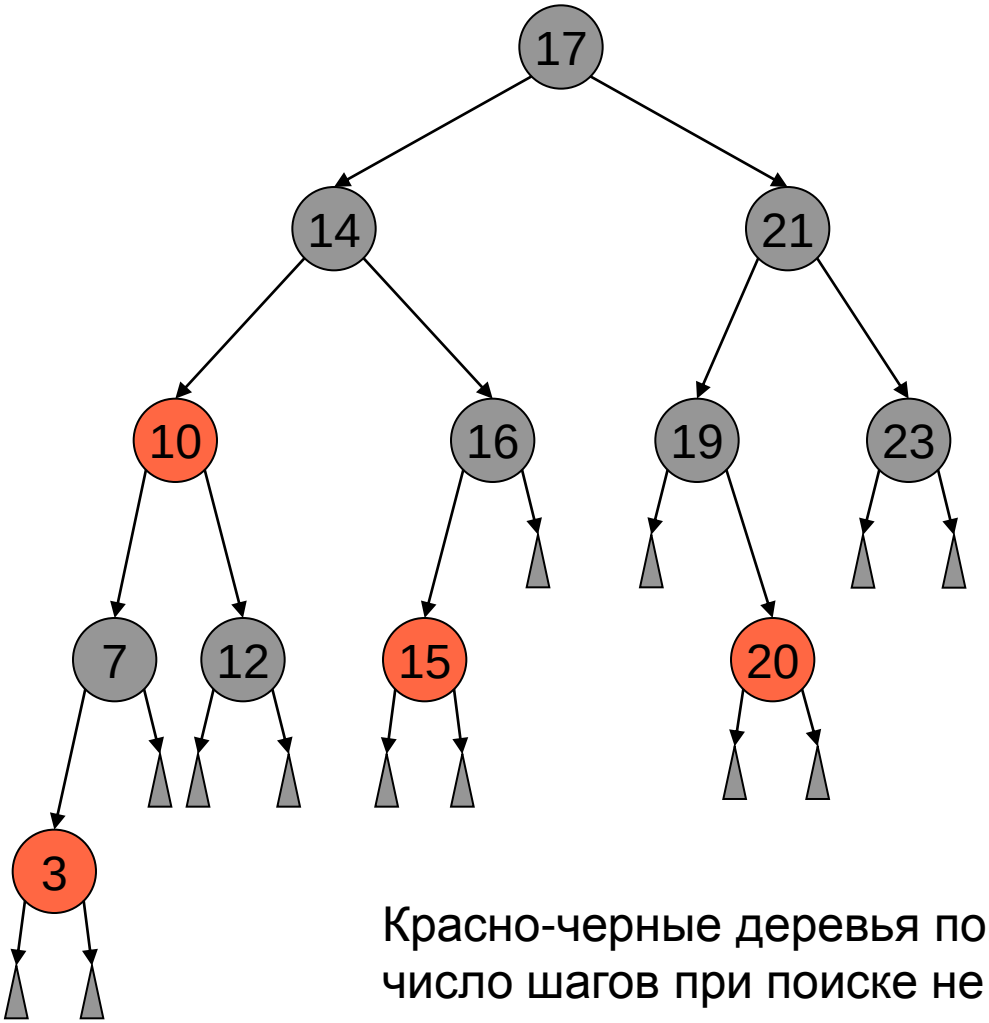


Красно-чёрные деревья (Red-Black Tree, RB-Tree)

Вставка



Красно-черные деревья



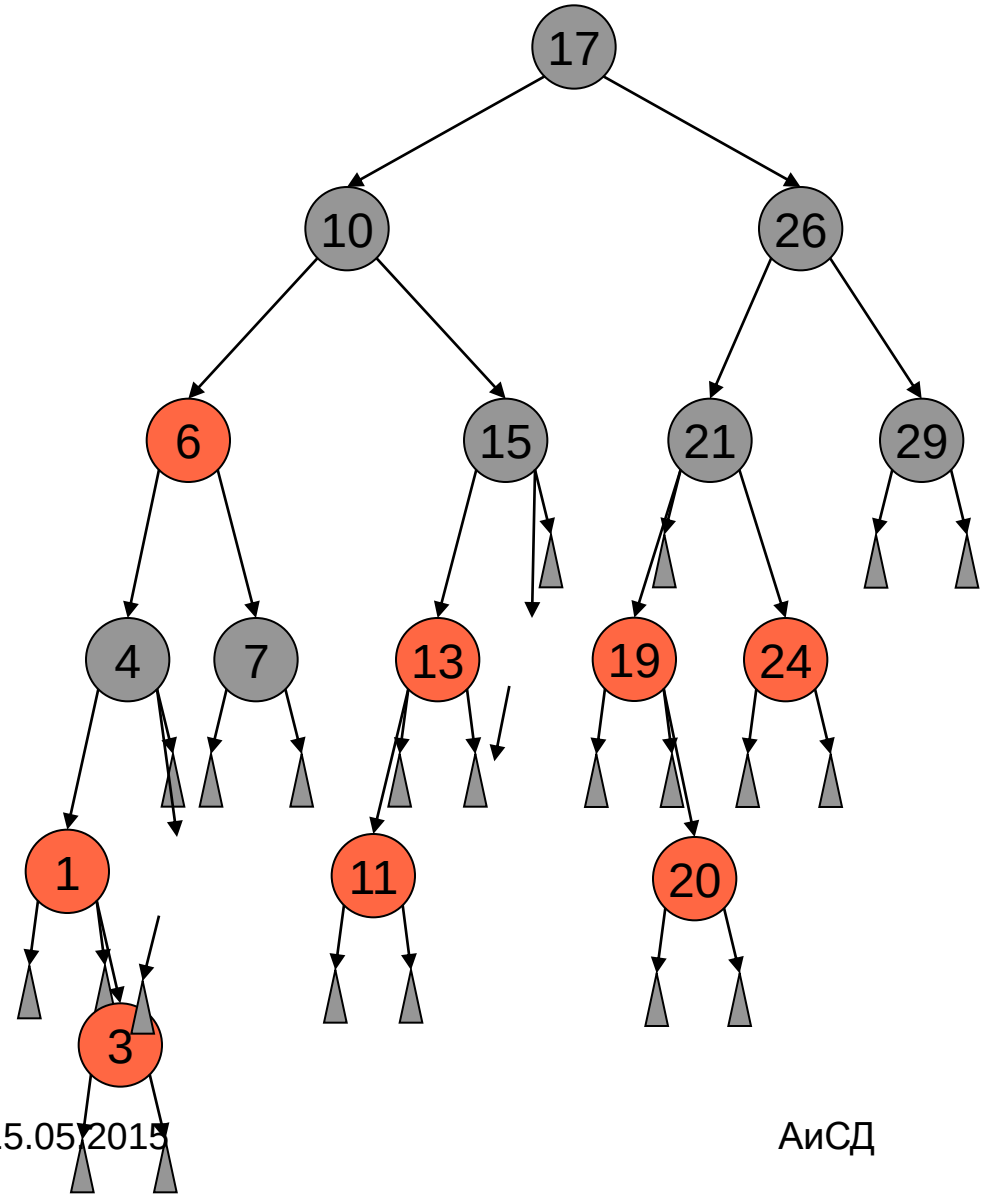
Свойства красно-черных деревьев

- Корень и пустые узлы всегда имеют черный цвет
- Красная вершина не может иметь красных потомков
- Максимальные «черные» длины путей, ведущих из корня к листьям (пустым узлам), одинаковы

Красно-черные деревья поиска достаточно эффективны: число шагов при поиске не превышает $2 \log_2 n$

Существуют эффективные процедуры добавления и удаления узлов, которые не нарушают свойств красно-черных деревьев

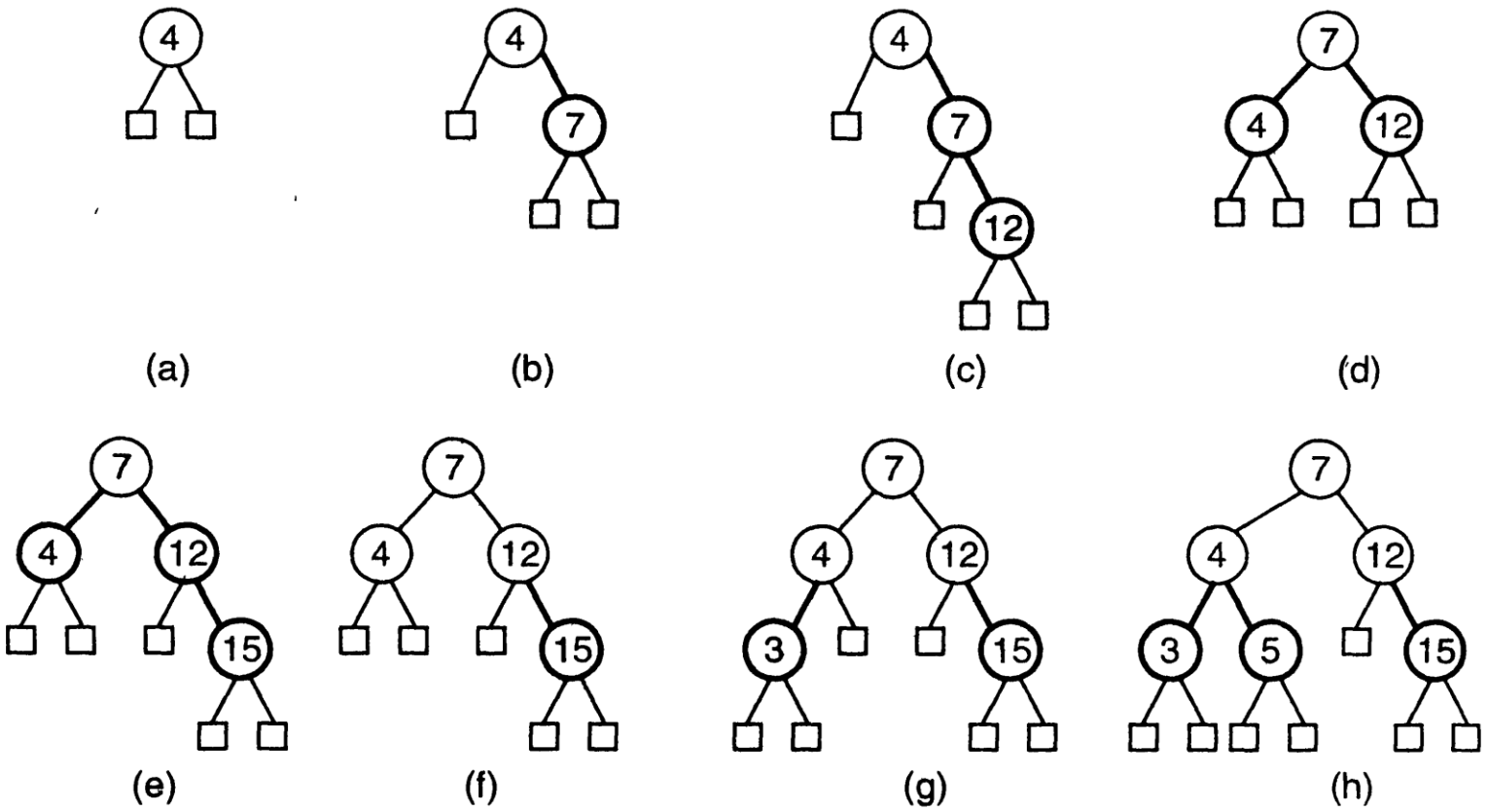
Алгоритм добавления ключа в красно-черное дерево



1. Добавляем ключ 19
2. Добавляем ключ 20
3. Добавляем ключ 11
4. Добавляем ключ 3…

Красно-чёрные деревья (Red-Black Tree, RB-Tree)

Создание дерева



Красно-чёрные деревья (Red-Black Tree, RB-Tree)

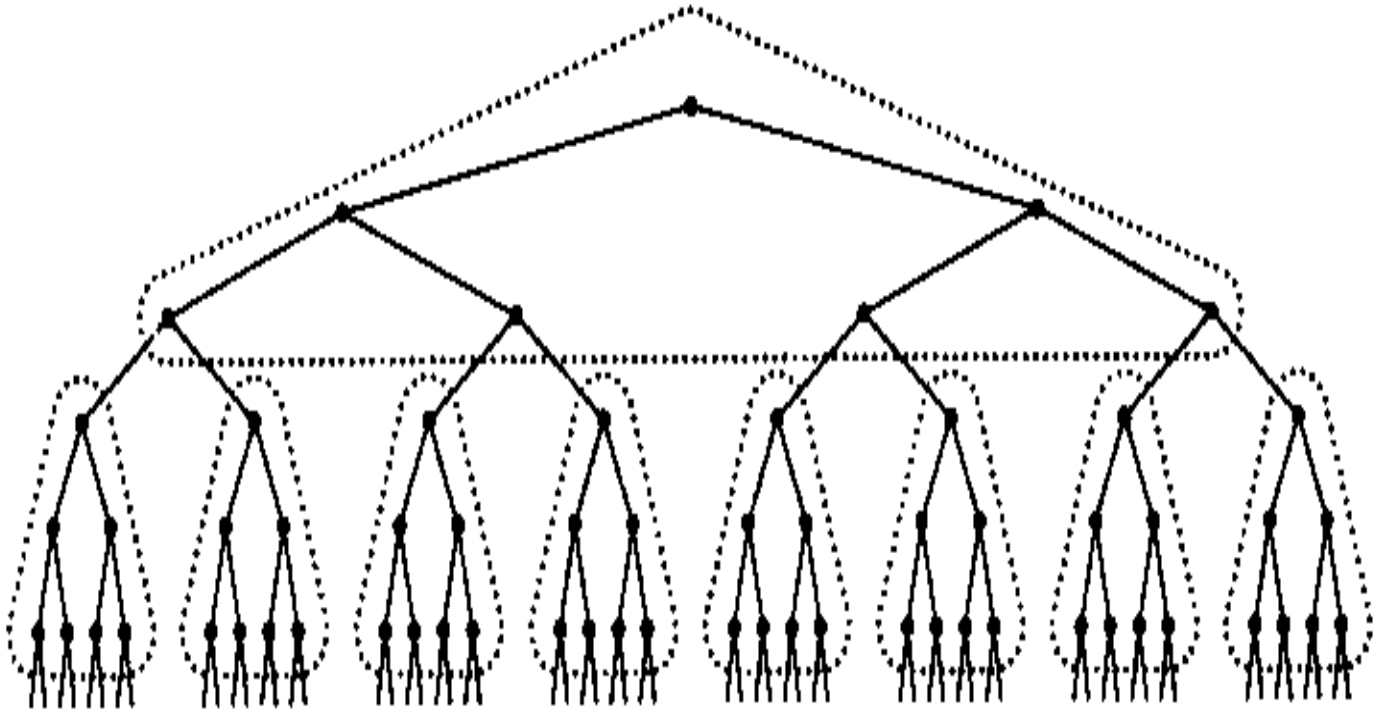
- Ввод объекта «ключ-элемент» в красно-черное дерево, хранящее n элементов, выполняется за O(log n) времени и требует максимум O(log n) перекрашиваний

http://rain.ifmo.ru/cat/view.php/vis/trees

Сильноветвящиеся деревья

Р. Бэйер (*R. Bayer*) и Е. МакКрейт (*E. McCreight*) 1970

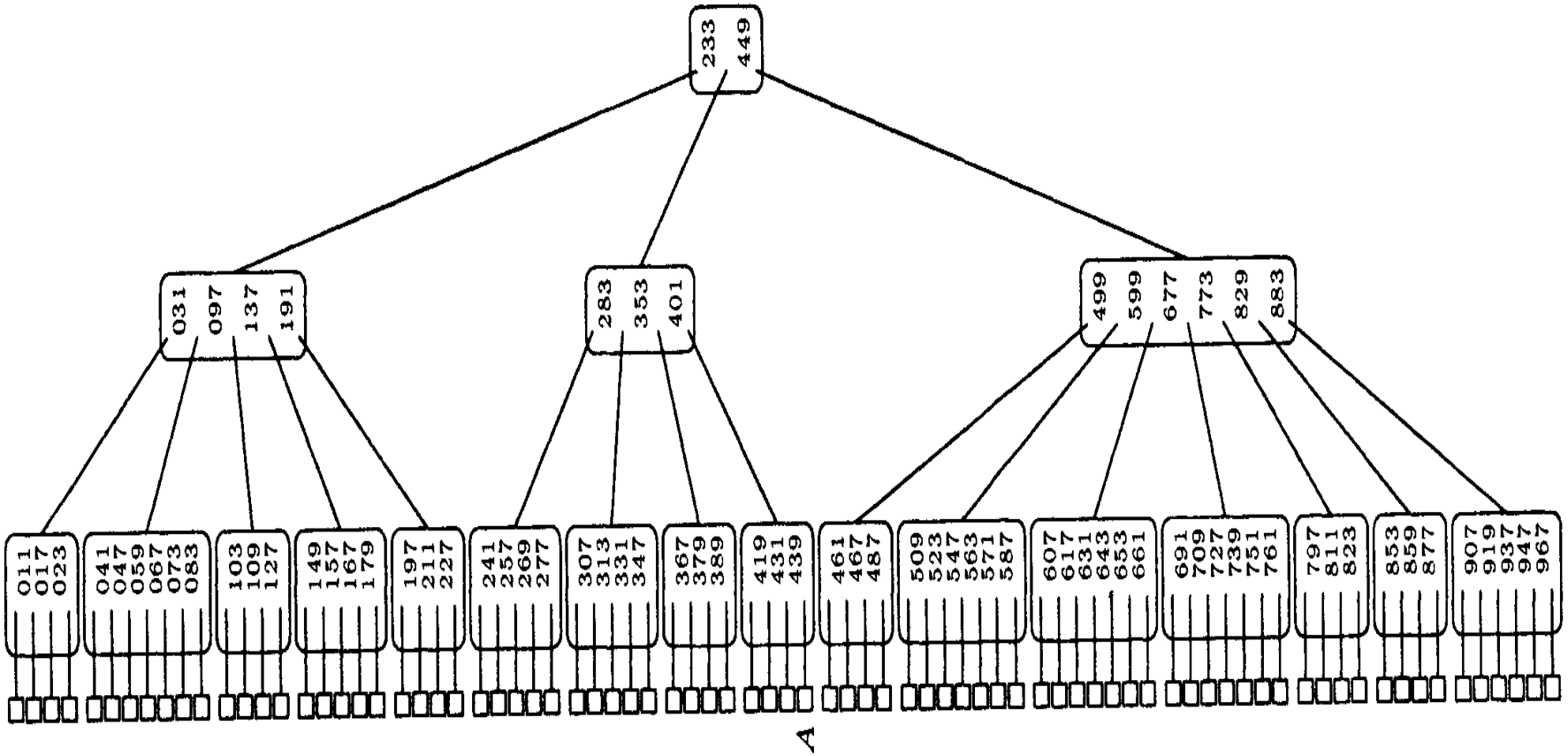
B-дерево порядка m=7



m=50-2000

B- деревья (B-trees, R.Bayer)

B-дерево порядка 7



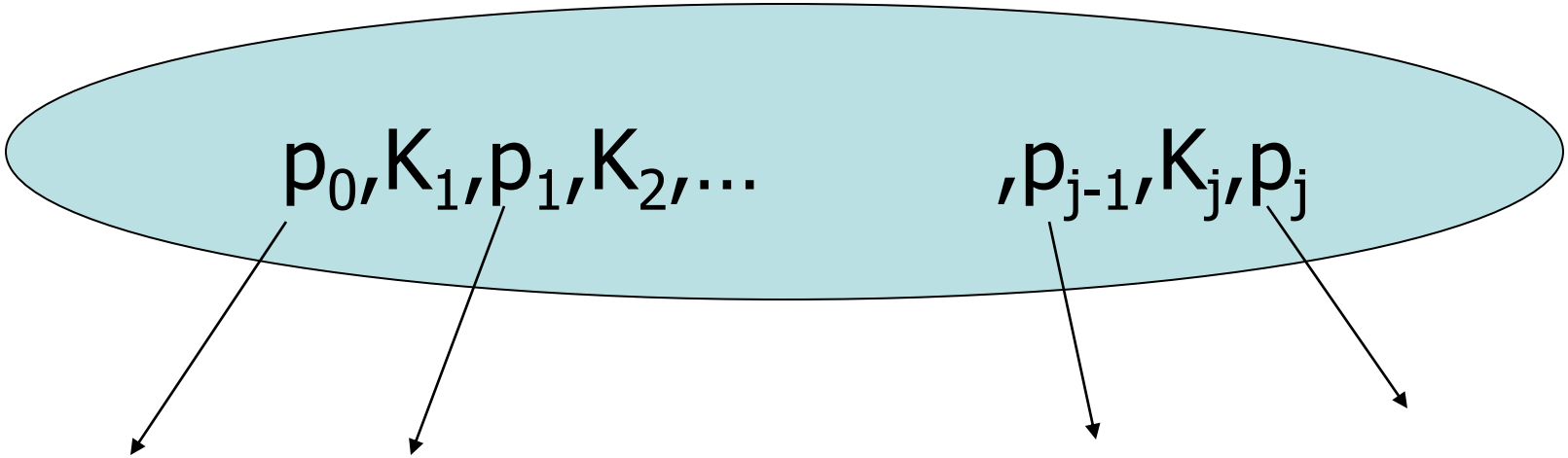
B- деревья (B-trees)

B-дерево m-го порядка

1. **Каждый узел имеет не более *m* потомков**
2. **Каждый узел, за исключением корневого узла и листьев, имеет не менее *m/2* потомков**
3. **Корневой узел, если он не является листом, имеет минимум двух потомков**
4. **Все листья находятся на одном и том же уровне и не содержат информации**
5. **Нелистовой узел с *k* потомками содержит *k* - 1 ключей.**

B- деревья (B-trees)

Узел B-дерева, содержащий j ключей и j+1 указатель



$$K_1 < K_2 < \ldots < K_j$$

Pi указывает на поддерево с ключами между K_i и K_{i+1}

B- деревья (B-trees)

типичный шаблон работы с объектом *x* *B- дерева* :

- ***x*** **← указатель на некоторый объект**
- **Disk_Read(x)**
- **Операции, обращающиеся и/или изменяющие поля *x***
- **Disk_Write(x) // Не требуется, если**
- **поля *x* не изменялись**
- **Прочие операции, не изменяющие поля *x***

B- деревья (B-trees)

1. **Каждый узел x содержит следующие поля:**

а) ***n (x)*, количество ключей, хранящихся в данный момент в узле *x*,**

б) **Собственно ключи, количество которых равно *n (x)* и которые хранятся в возрастающем порядке**

$$key_1(x) \le key_2(x) \le \bullet\bullet\bullet \le key_{n(x)}(x)$$

в) **Логическое значение *leaf (x)*, равное true, если *x* является листом, и false, если *x* — внутренний узел.**

B- деревья (B-trees)

2. **Кроме того, каждый внутренний узел x содержит $n\,(x)+1$ указателей**

$$c_1(x),\ c_2(x),\ \ldots\ c_{n\,(x)+1}(x)$$

на дочерние узлы. Листья не имеют дочерних узлов.

B- деревья (B-trees)

3. Ключи $key_i\ (x)$ разделяют поддиапазоны ключей, хранящихся в поддеревьях:
если k_i — произвольный ключ, хранящийся в поддереве с корнем $c_i(x)$ то

$$k_1 \le key_1\ (x) \le\ k_2 \le\ key_2\ (x)\ \ \ldots \le$$

$$key_{n(x)}\ (x) \le\ k_{n(x)+1}$$

B- деревья (B-trees)

4. **Все листья расположены на одной и той же глубине, которая равна высоте дерева h.**
5. **Имеются нижняя и верхняя границы количества ключей, которые могут содержаться в узле.**

Эти границы могут быть выражены с помощью одного фиксированного целого числа $t \geq 2$, называемого *минимальной степенью* (minimum degree) B-дерева:

B- деревья (B-trees)

- **Каждый узел, кроме корневого, должен содержать как минимум $t - 1$ ключей.**

  Каждый внутренний узел, не являющийся корневым, имеет,таким образом, как минимум t дочерних узлов.

  Если дерево не является пустым, корень должен содержать как минимум один ключ.

- **Каждый узел содержит не более $2t — 1$ ключей. Таким образом, внутренний узел имеет не более $2t$ дочерних узлов. Узел *заполнен* (full), если он содержит ровно $2t — 1$ ключей.**

B- деревья (B-trees)

- **Высота B-дерева с $n \geq 1$ узлами и степенью $t \geq 2$ не превышает**

$$\log_t (n+1)/2$$

B- деревья (B-trees)

Алгоритм поиска:

BTreeSearch (x, k);

i←1;

While **i≤** ***n (x)*** **&** ***k > key$_i$(x)***

i ← i +1;

If **i≤** ***n (x)*** **&** ***k = key$_i$(x)***

then return ***(x,i);***

If ***leaf (x)***

then return ***null***

else ***DiskRead(c$_i$(x));*** **return**

BTreeSearch (*c$_i$ (x)*, k);

B- деревья (B-trees)

Количество дисковых страниц, к которым обращается BTreeSearch

$$O(h) = O(\log_t n)$$

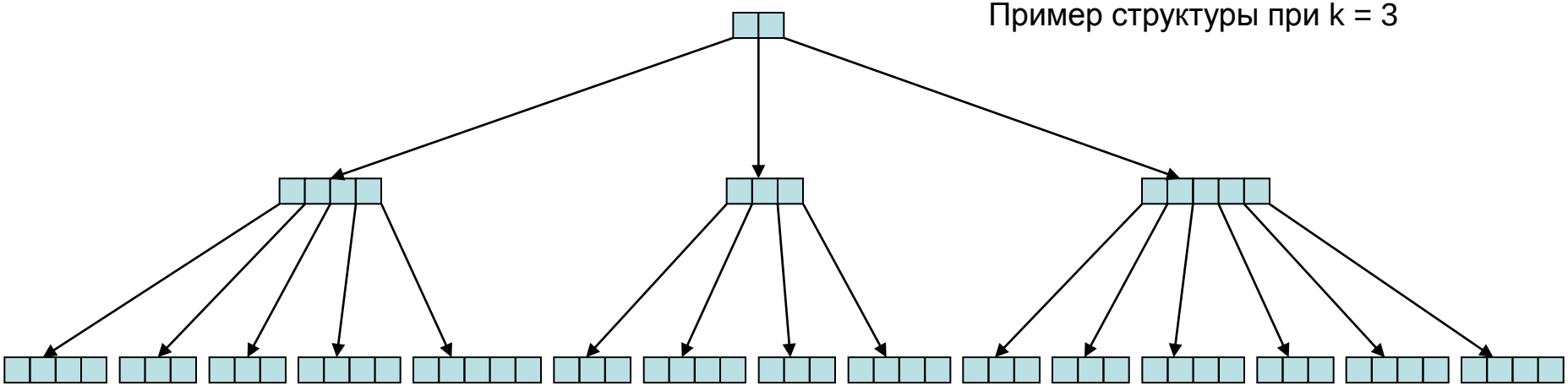
Общее время вычислений

$$O(t*h) = O(t*\log_t n)$$

B- деревья: создание пустого дерева

```
BTreeCreate( );
x ← AllocateNode ( );
leaf(x) ← true;
n(x) ← 0;
DiskWrite (x);
root ← x
```

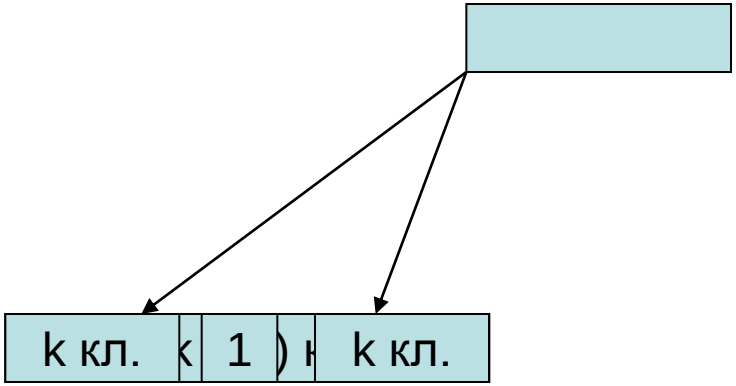
Обобщение 2-3-дерева – B-дерево k-го порядка



Структура B-дерева:

- Корневой узел содержит от 1 до 2*k ключей
- Прочие узлы содержат от k до 2*k ключей
- Ключи упорядочены (возможен быстрый поиск)
- Промежуточные узлы имеют ***все*** ссылки
  (корень – от 2, остальные – от (k+1) до (2*k+1) ссылки)
- Все терминальные узлы (листья) находятся на одном уровне

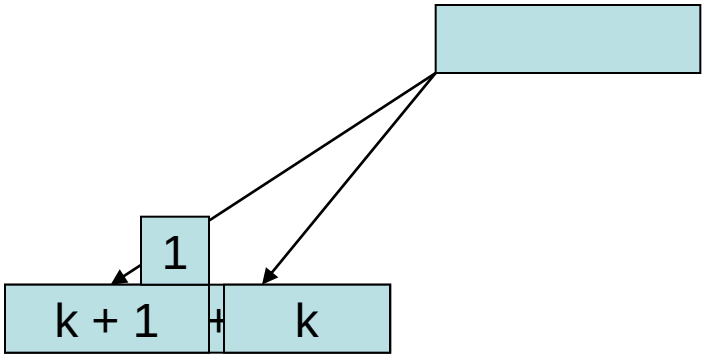
Расщепление узла B-дерева k-го порядка при вставке ключа



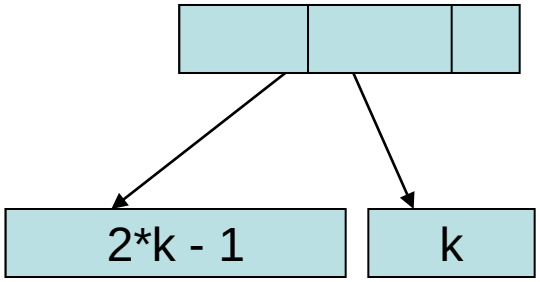
1. При вставке ключа в терминальный узел образовалось переполнение узла
2. Делим узел на 3 узла: k, 1 и k ключей
3. Перемещаем средний ключ на предыдущий уровень

Модифицированное B-дерево (B^+-дерево)

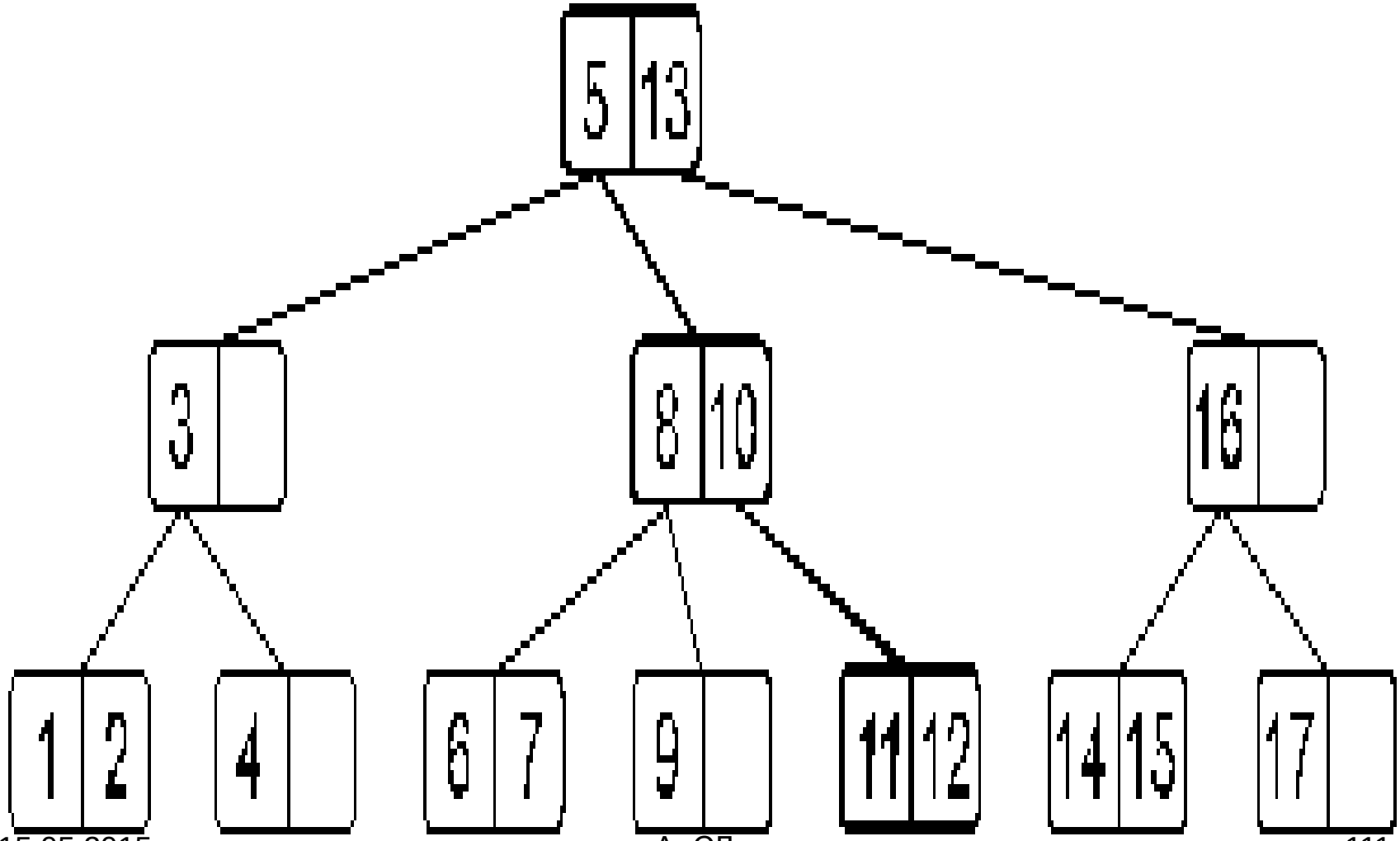
1. Только терминальные узлы содержат ссылки на данные; промежуточные узлы служат только для поиска информации.
2. При расщеплении узла происходит создание копии ключа:



3. При слиянии узлов ключ, пришедший с более высокого уровня, уничтожается:



B- деревья: ПОИСК

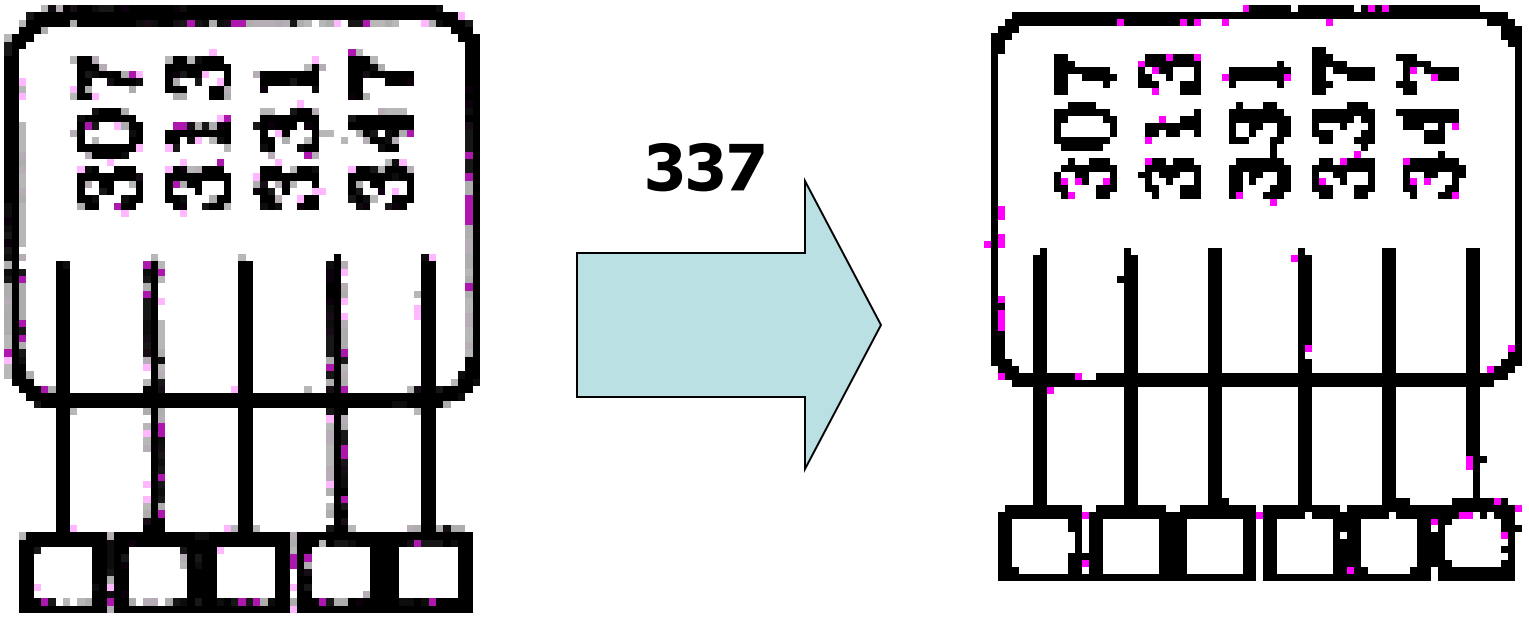


B- деревья: добавление

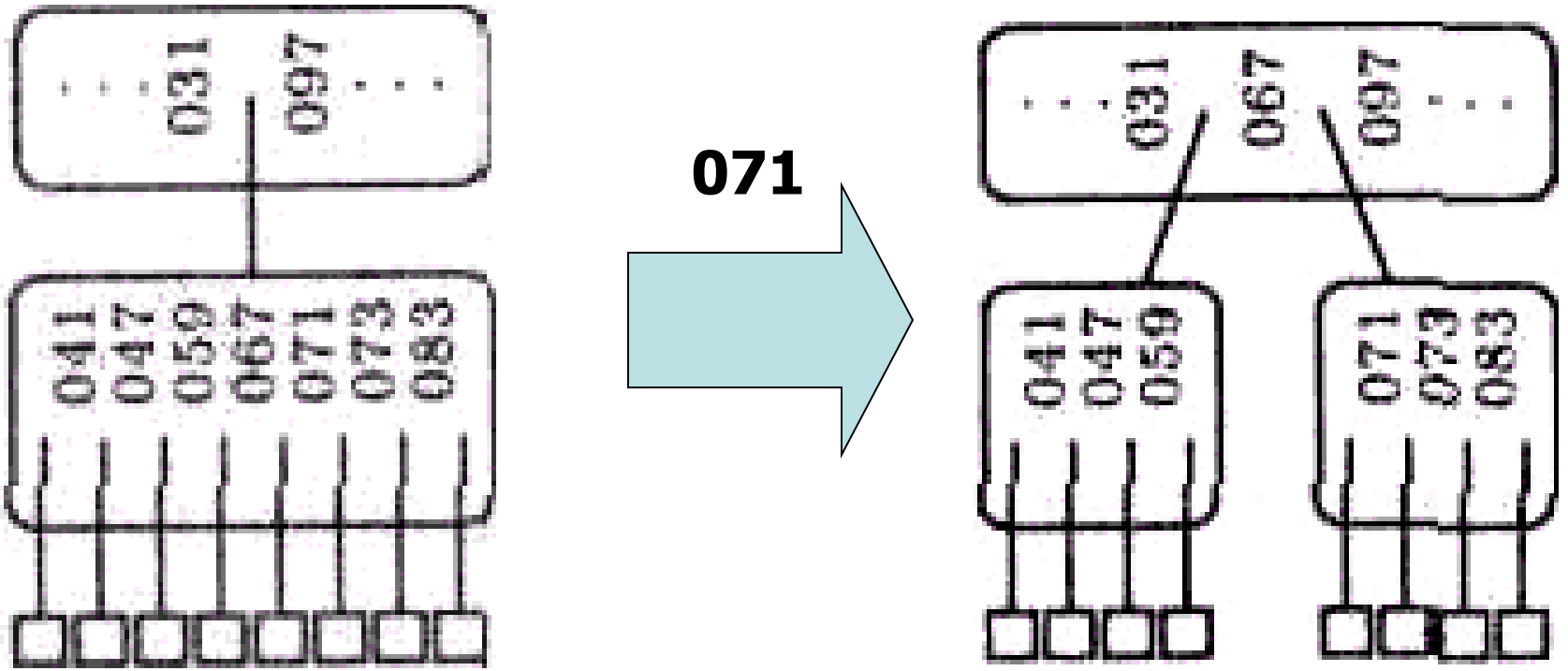
1. Поиск листового узла Node, в который следует произвести добавление элемента Item.
2. Добавление элемента Item в узел Node.
3. Если Node содержит больше, чем NumberOfItems элементов (произошло переполнение), то

- делим Node на две части, не включая в них средний элемент;
- Item=средний элемент Node;
- Node=Node.PreviousNode;
- Переходим к пункту 2.

B- деревья: добавление



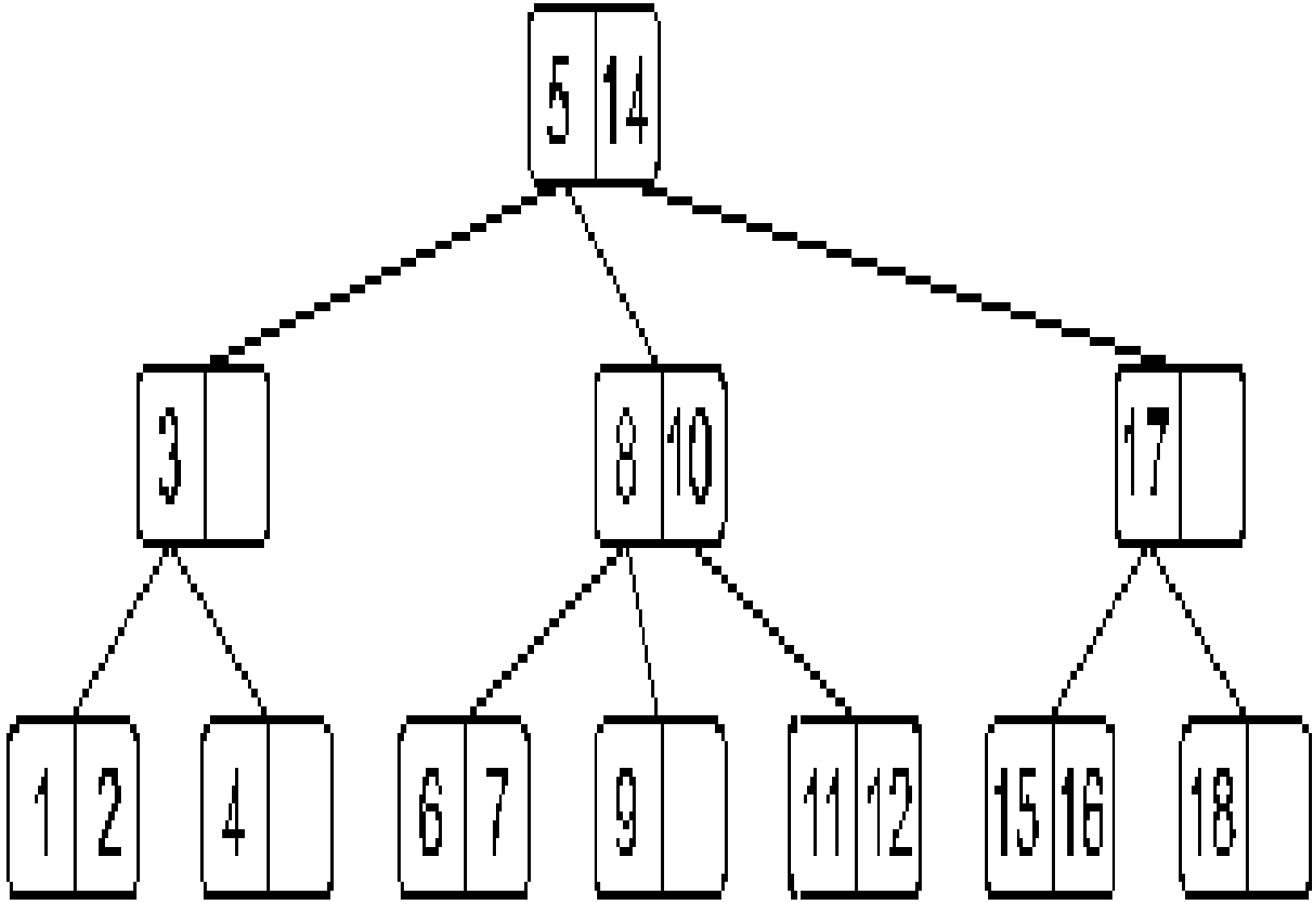
B- деревья: добавление



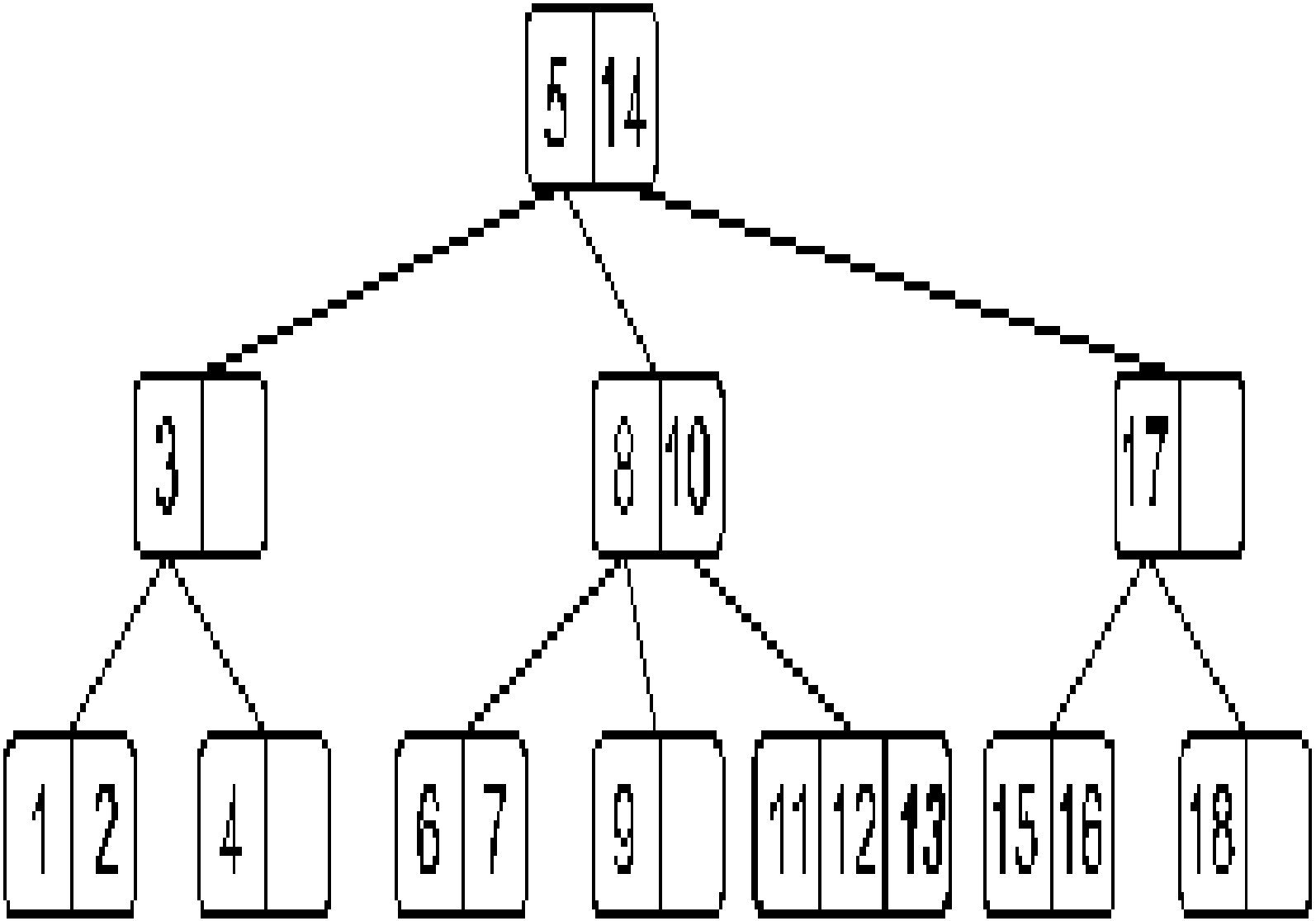
B- деревья: добавление

```
BTreeInsert(T,k);
r ← root(T);
if n(r) = 2t-1;
    then      s ← AllocateNode ( );
              root(T) ← s;
              leaf (s) ← false;
              n (s) ← 0;
              c1(s) ← r;
              BTreeSplitChild(s,1,r);
              BtreeInsertNonfull(s,k);
    else  BtreeInsertNonfull(r,k);
```

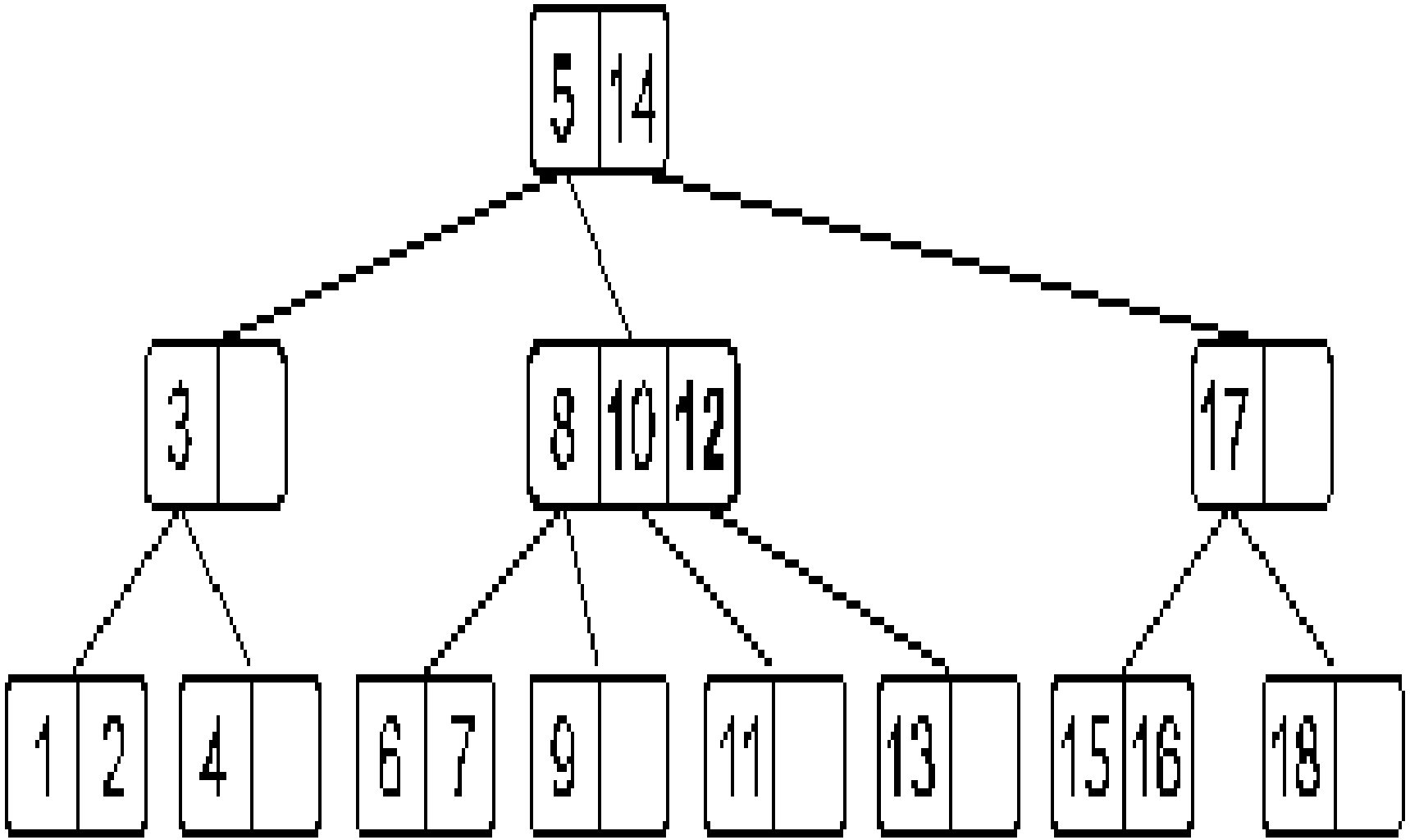
B- деревья: добавление



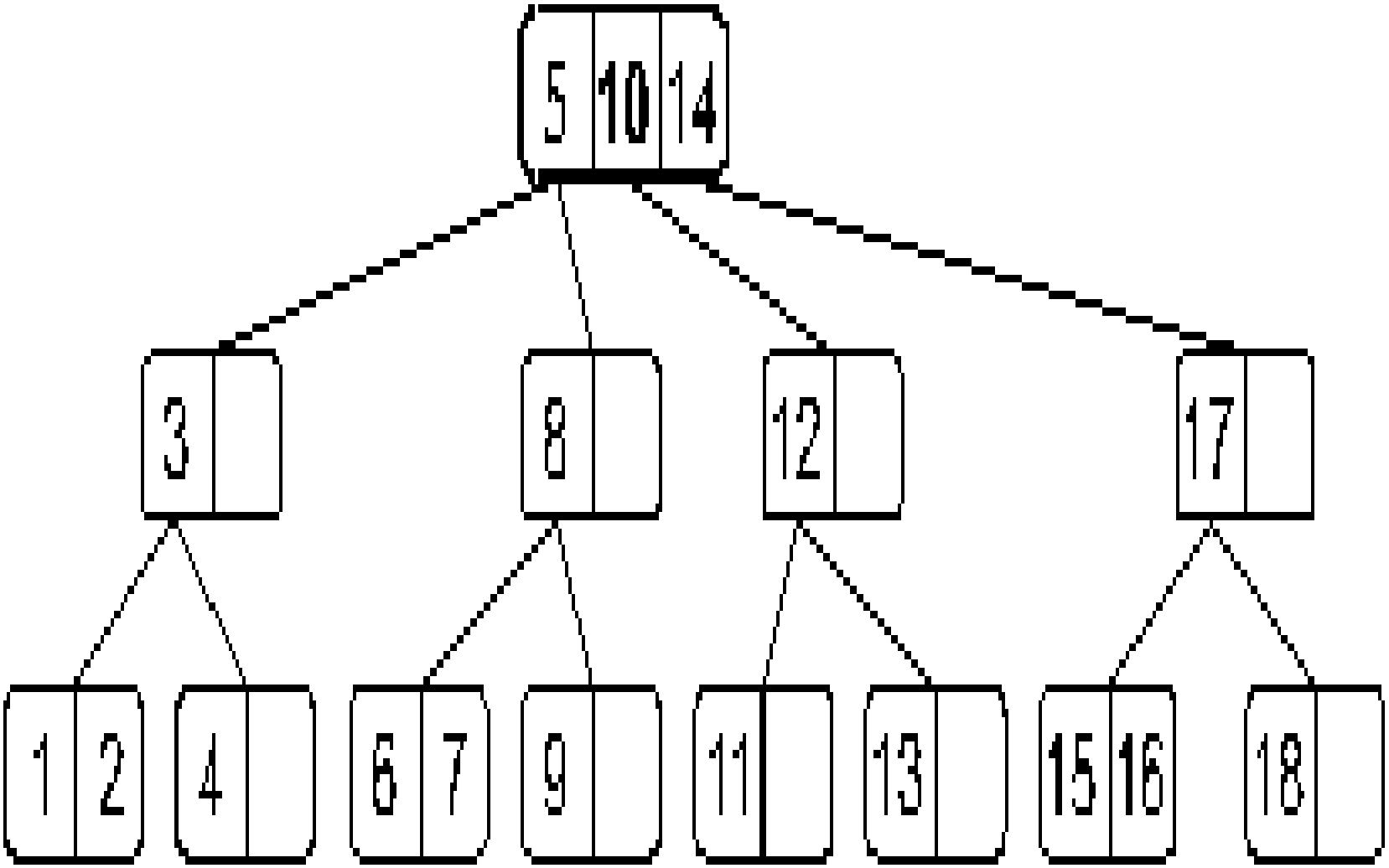
B- деревья: добавление



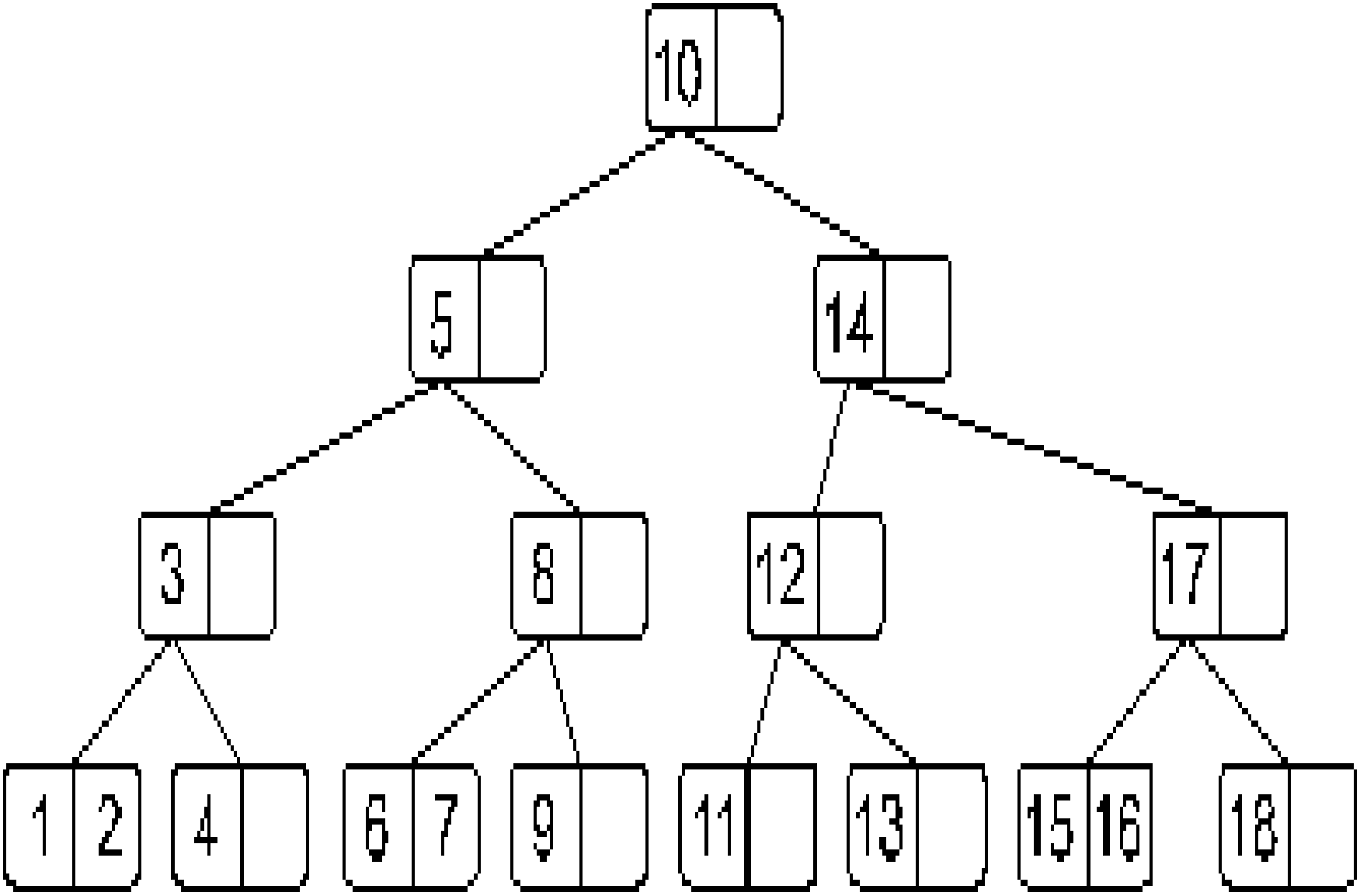
B- деревья: добавление



B- деревья: добавление



B- деревья: добавление



B- деревья: удаление

1. Если узел k находится в узле x и x – лист удаляем k из x
2. Если узел k находится в узле x и x – внутренний узел

- Если дочерний по отношению к x узел y, предшествующий ключу k, содержит не менее t ключей, то находим k^1 – предшественника k в поддереве, корнем которого является y,рекурсивно удаляем k^1 и заменяем k на k^1

B- деревья: удаление

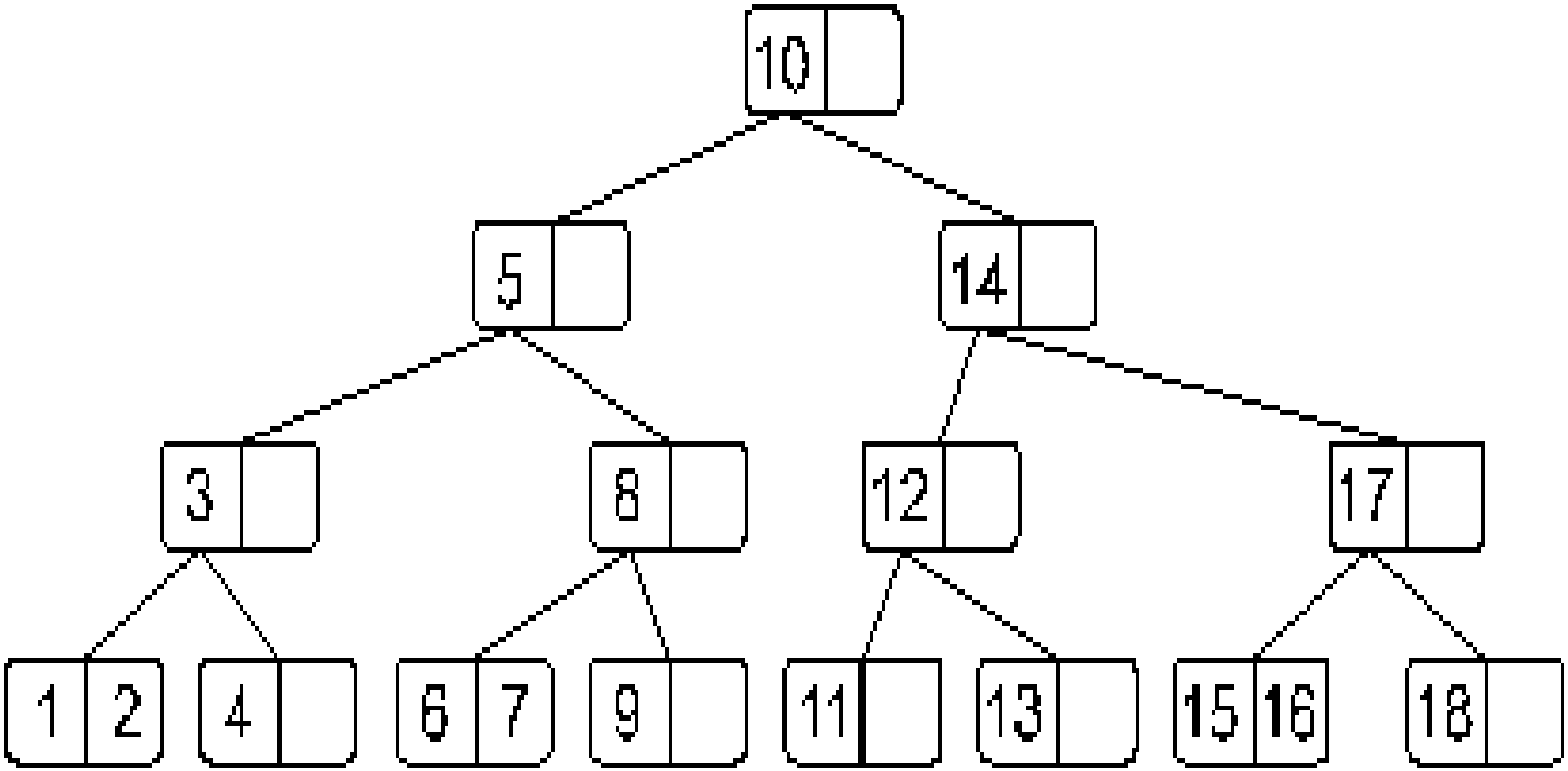
- Если дочерний по отношению к x узел z, следующий за ключом k, содержит не менее t ключей, то находим k^1 – следующий за k в поддереве, корнем которого является z, рекурсивно удаляем k^1 и заменяем k на k^1

B- деревья: удаление

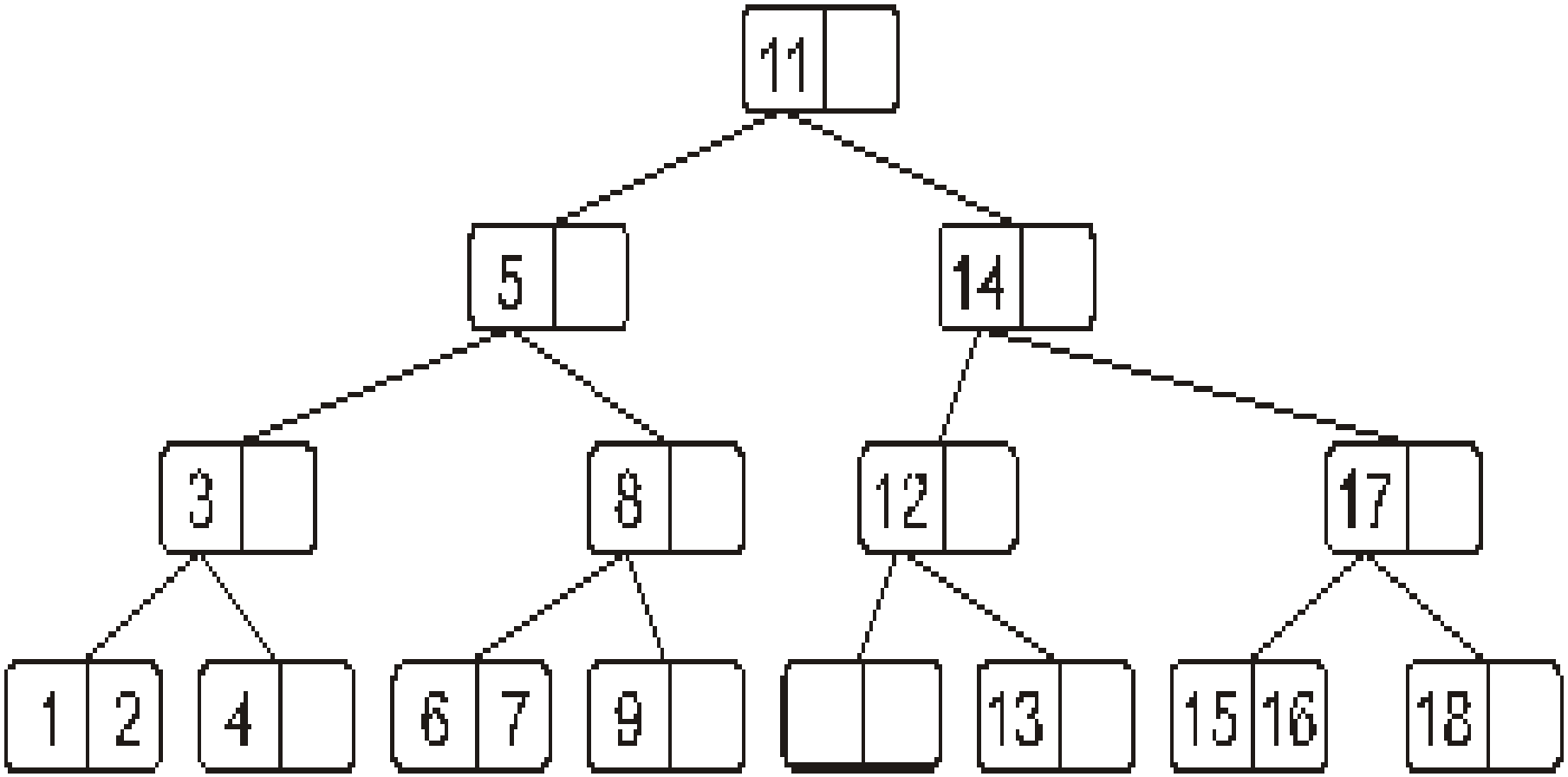
- если y и z содержат по t-1 ключей, то осуществляется слияние z в y

из x удаляется k и ссылка на z, y содержит 2t-1 узел

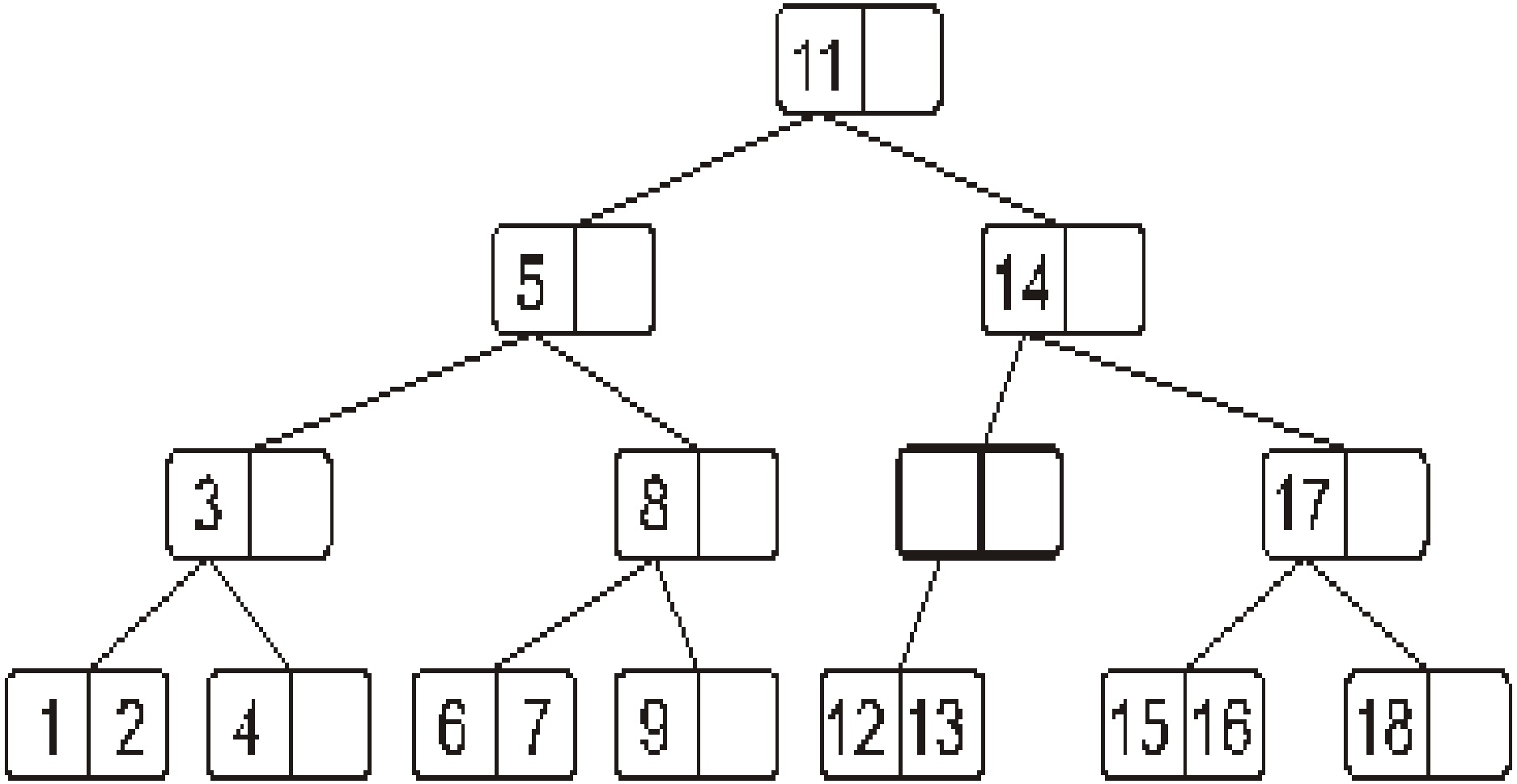
B- деревья: удаление



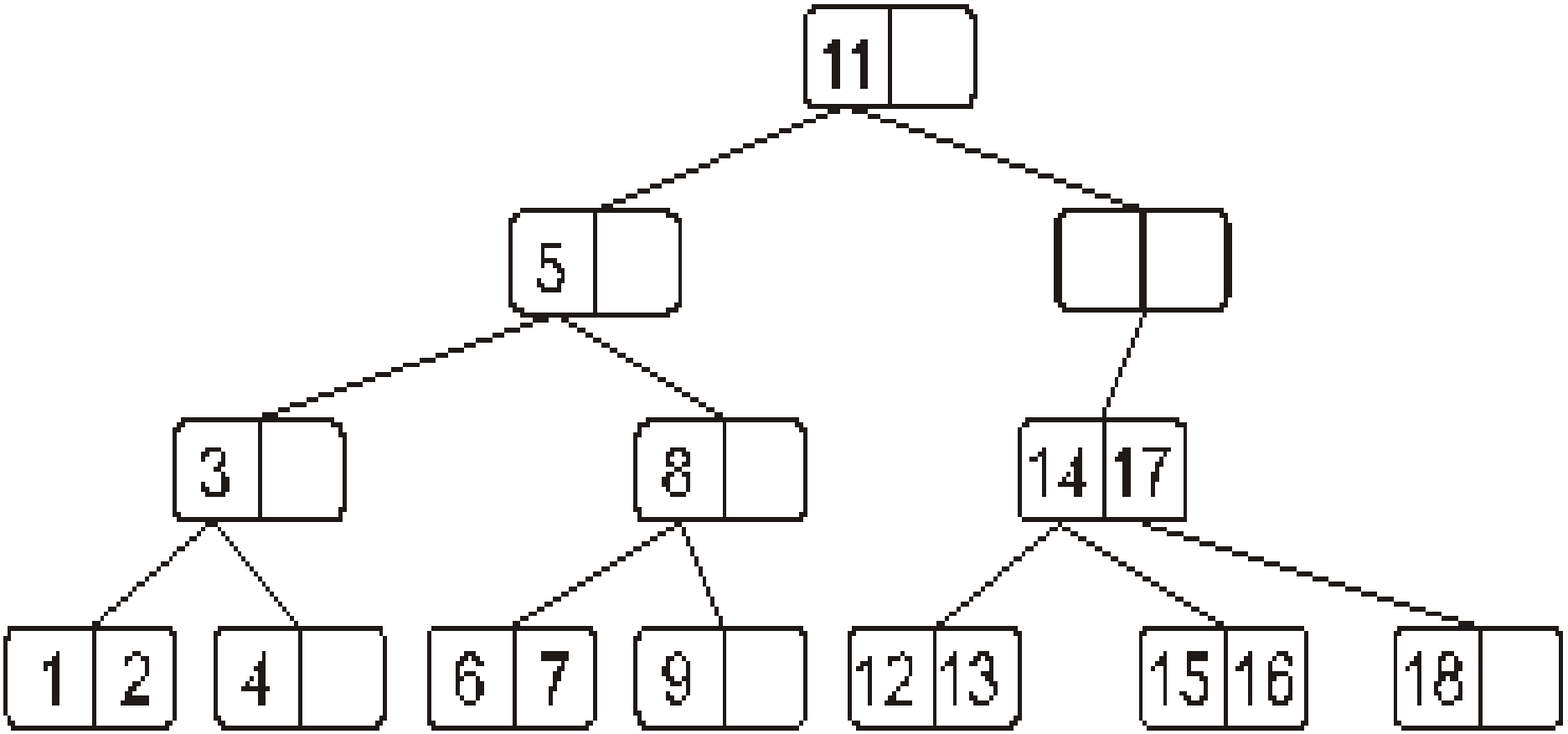
B- деревья: удаление



B- деревья: удаление



B- деревья: удаление



B- деревья: удаление